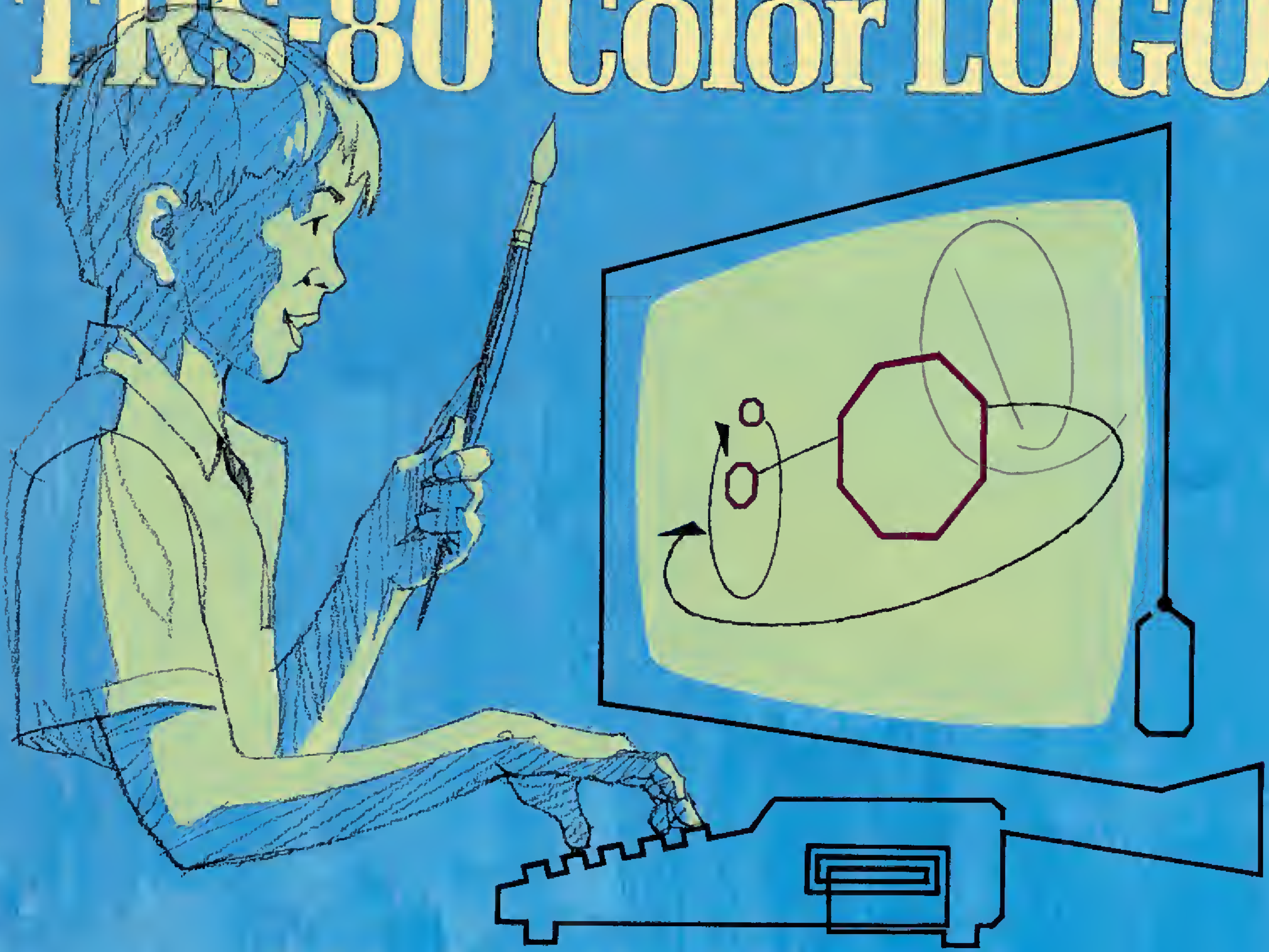
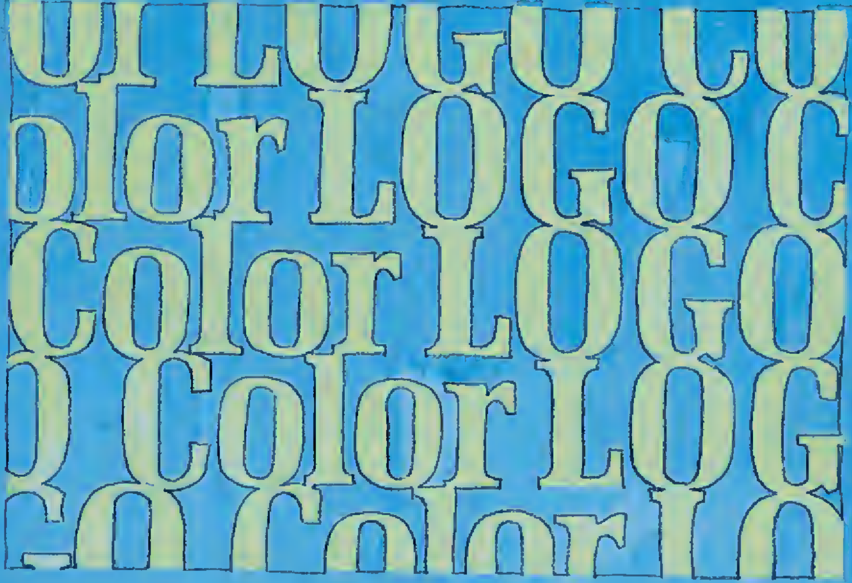
Cat. No. 26 - 2722

TRS-80® Color LOGO





Radio Shack™
TRS-80
COLOR COMPUTER

THIS WARRANTY SUPERSEDES ALL PRIOR WARRANTIES

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement The original CUSTOMER's sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items

B RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D Except as provided herein, <u>RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.</u>

E Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER

III LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE" IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs

D Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on <u>one</u> computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software

C CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on <u>one</u> computer and as is specifically provided in this Software License Customer is expressly prohibited from disassembling the Software.

E CUSTOMER is permitted to make additional copies of the Software <u>only</u> for backup or archival purposes or if additional copies are required in the operation of <u>one</u> computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G All copyright notices shall be retained on all copies of the Software

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK

VI STATE LAW RIGHTS

The warranties granted herein give the <u>original</u> CUSTOMER specific legal rights, and the <u>original</u> CUSTOMER may have other rights which vary from state to state

Radio Shack® Color LOGO

By

Larry Kheriaty

and

George Gerhold

Foreword

For more than a decade, the authors have been involved in the use of computers in education, and particularly with Computer-Assisted Instruction. Our experience made us aware of the problems of getting students started right in programming and of LOGO's potential to help solve that problem, and we determined that it would be worthwhile to develop a version of LOGO which ran on low-cost hardware and allowed relatively long sets of procedures. A review of the features of other versions of LOGO revealed that list processing features were not used in most applications. List processing was therefore eliminated, allowing the design objectives to be met and allowing the addition of some features, notably multiple turtles.

Our debt to the original designers of the LOGO language is acknowledged in Chapter 1. We would also like to thank our children Aaron, Jenell, and Kirstin, whose responses to early versions of Color LOGO convinced us that we were on the right track.

George Gerhold

George Gerhold

Larry Kheriaty

Larry Kheriaty

Table of Contents

INTRODUCTION

Radio Shack® Color LOGO is an educational computer language. The language can be used to draw pictures on the computer's video display, using a shape on the screen called a "turtle."

Color LOGO is designed to let children learn by exploring. Children plan an action, then enter simple commands that move the turtle forward or back, or turn it in any direction. Here are a few of the special features of Color LOGO:

- Line-oriented editing allows you to write and save sequences of turtle movements (called "procedures").
- A "doodle mode" lets children who are too young to read or type accurately use the program.
- A "SLOW" command lets you control how fast the turtle moves.
- Screen colors can be changed.
- Limited animation is possible with Color LOGO.
- Variables and arithmetic expressions can be used in the sets of turtle movements that you write and save.

Color LOGO is a language for beginners. For this reason, the Color LOGO manual has been written to guide you through use of the language, step by step, with many examples and illustrations. Here is a summary of the organization of the manual:

1. Chapters 1 through 10 introduce turtle graphics and the LOGO syntax. Readers who are already familiar with LOGO may wish to skim these chapters or to bypass them in favor of the language summary in Appendix 1.

2. Chapters 11 through 17 cover features unique to Color LOGO.

    a. Chapters 11 through 13 provide hints for using Color LOGO with very young children.

    b. Chapters 14 through 16 introduce the use of multiple turtles and new turtle shapes.

    c. Chapter 17 contains sample sets of turtle movements that you may wish to explore.

This Radio Shack Color LOGO package can be used with a ROM-based TRS-80 Color Computer with Color BASIC and with at least 16K of memory. Procedures that you write can be saved on cassette tape.

1. A BIT ABOUT COLOR LOGO

Color LOGO is a computer language for children. Like all the best things for children, Color LOGO can provide endless fascination and challenge for adults as well. At first glance Color LOGO may seem to be simply a language for drawing pictures, since the result of running a Color LOGO program is almost always a picture. However, Color LOGO is far more than an easy way to draw pictures. Color LOGO is a tool for learning about some of the most powerful concepts in mathematics, physical sciences, computer science, and problem solving—but in a way so appealing and simple that "even a kid can do it."

Notice that we said that Color LOGO is a language for learning; we very intentionally did not say that Color LOGO is a language for teaching. The role of the learner is all important. Color LOGO puts the student in the role of explorer, one who sets goals (problems to solve) and tries to find a way to those goals. The role of the teacher is guide, one who stays in the background as much as possible, one who does not set the goals for the learner, and one who only assists when asked. Effective use of Color LOGO has much of the flavor of play: "It's not whether you win or lose, but how you play the game." The goal the student reaches is not as important as the process of seeking the goal.

Color LOGO is based on a set of ideas for use of the computer first developed under the name LOGO. Many people have contributed to the LOGO project, too many to list, but we must mention the names Wallace Feurzeig, Harold Abelson, Andrea diSessa, and—with special emphasis—Seymour Papert. Most of the development and testing of LOGO was done at MIT. There were two vital steps in bringing the LOGO approach to the attention of the educational community. One was the publication of two books; *Mindstorms* by Papert and *Turtle Geometry* by Abelson and diSessa. Any serious user of LOGO will want to read those books. The other was the implementation of the LOGO language on microcomputers, a step which decisively moved LOGO from the laboratory into the classroom. If you are already familiar with LOGO, you will find much of Color LOGO to be familiar too. Wherever possible we have kept the same syntax as LOGO, and the logical structures of the two languages are essentially the same. Most of the graphics programs in books on LOGO will run in Color LOGO without change.

Color LOGO is not just LOGO under another name for another computer; there are some very important differences between the two. Color LOGO is a graphics language, but LOGO also includes a number of string operations drawn from the LISP language. Color LOGO uses strings only as labels on graphics. Color LOGO provides multiple turtles, whereas most versions of LOGO provide only a single turtle. Color LOGO thus can be used to introduce important concepts like multi-programming and messages via mail boxes, but still with great simplicity. Consequences of multiple turtles include provision for simple animation and the potential for user-created games. All these are possible because, in contrast to LOGO, the memory requirements of Color LOGO are modest. Color LOGO also provides a mode for doodling, designed for children who are too young to type keywords reliably.

If you are just starting on computers, all that sounds rather complex. That's because we're just talking about it instead of doing it. Let's do it.

2. GETTING STARTED

There are two ways you can start Color LOGO running on your TRS-80 Color Computer: from a plug-in cartridge or from disk. You will want to have some way to store your favorite creations for future display, so you will want to have either a disk drive or a cassette player attached to your computer system. Consult the chapters on "Installation" and "Operation" in your copy of the *TRS-80 Color Computer Operation Manual* for instructions as to proper cable connections for the disk and/or the cassette player. DO NOT TURN ON THE POWER YET!

NOTE: If you have the ROM version of Color LOGO (Cat. No. 26-2722), follow the steps under "Plug-In ROM Cartridge." Then turn the page to "Using the Color LOGO Program." For the disk version (Cat. No. 26-2721), follow the steps under "Disk," then under "Using the Color LOGO Program."

Plug-In ROM Cartridge:

Plug the Color LOGO cartridge into the slot on the right side of your TRS-80 computer. Check that the label is up and that the cartridge is seated firmly. Then turn the power on. (The power switch is on the back left corner of the computer.) The screen should display the prompt

```
    COLOR LOGO COPYRIGHT 1982
LARRY KHERIATY & GEORGE GERHOLD
      ALL RIGHTS RESERVED.
     LICENSED TO TANDY CORP.

LOGO:
```

Disk:

Plug the Color Computer Disk Controller into the slot on the right side of the computer. Then turn the system on. (The computer power switch is on the back left corner of the computer. The disk drive switch is on the back of the drive, upper left corner.) You should see the prompt

```
DISK EXTENDED COLOR BASIC 1.0
COPYRIGHT © 1981 BY TANDY
UNDER LICENSE FROM MICROSOFT

OK
```

Follow these steps:

1. Insert the Color LOGO diskette with the square notch up and the label facing right, into Drive 0 (the drive closest to the Color Computer on the cable).

2. Close the disk drive latch.

3. Type L O A D M ␣ " L O G O " and press ENTER.

4. After the light on the disk drive goes out, type E X E C and press ENTER.

   The system will now display the prompt shown above under **"Plug-In ROM Cartridge."**

5. Remove the Color LOGO diskette from the disk drive.

Using the Color LOGO Program:

Whichever way you started, you should now be in BREAK mode, which is indicated by the prompt which ends

```
LOGO:
```

at the left of the screen. You can return to BREAK mode at any time by any one of three actions:

1. BREAK key will interrupt whatever you are doing and return to BREAK mode

2. Reset button (located on the right rear corner of the computer) will always return you to BREAK mode, but you will lose all programs in memory

3. A complete restart (as described above) will place you in BREAK mode.

BREAK mode will be covered in detail in Chapters 3 and 9; for now let's move into RUN mode by pressing R. There is the turtle, sitting in the center of the screen facing straight up. Admittedly this turtle does not bear a strong resemblance to the ordinary pond-type turtle, but—like an ordinary turtle—it can crawl forwards and backwards, it can turn right and left. Unlike ordinary turtles, computerized turtles can drag their tails to leave tracks (in colors) or raise their tails and not leave tracks. Turtles can even be made invisible.

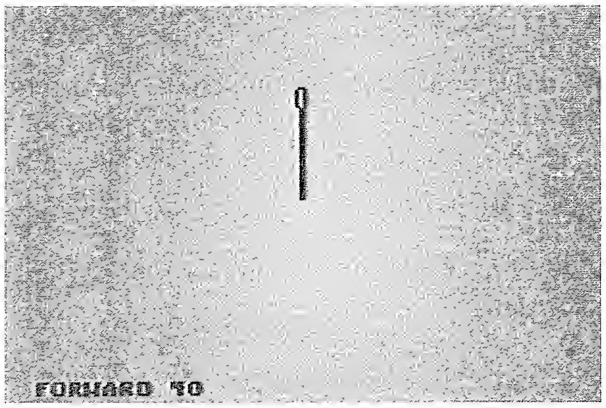
The name turtle was given originally to a tiny mechanical robot which could be made to crawl around the floor under computer control. The name probably had much more to do with the speed of the robot than with the shape of the robot. The track left by the turtle was called a "turtle graphic." The term turtle graphics is now used to indicate a way of drawing where lines are described by a direction and a length (the alternative is to describe a line by giving the coordinates of the two end points of the line, a method called—strangely—vector graphics). The item which moves is called the turtle, even when it is just a shape on the screen. Color LOGO is a language for controlling turtles.

We have a turtle in the center of the screen, itching for action. Let's tell the turtle to move forward. Simply type*

```
FORWARD 40
```

*Commands or program lines which you are to enter into the computer will be represented by the typeface which you see in "**FORWARD 40**" above. (Some messages from the computer will also appear in this typeface, but the context will make the difference clear.)

Then press $\boxed{\text{ENTER}}$.* The number tells the turtle how far forward to move. After you type **FORWARD 40** the screen will show



```
FORWARD 40
```

See the turtle track?

It won't be long before you get tired of typing **FORWARD** all the time, so there is an abbreviation which has the same effect. Enter the following (type it and press $\boxed{\text{ENTER}}$):

```
FD 10
```

Try to get a feel for the screen size and resolution. Try

```
FD 1
```

It's almost too little to see. Then try with a larger number, like

```
FD 100
```

The turtle moved, but it didn't leave a track. When the turtle goes off the top of the screen, it reenters at the bottom, a process which is called "wrapping around." A turtle can wrap around only by temporarily raising its tail, so no track appears for any step where the turtle wraps around.

*NOTE: If you forget to leave a space between "**FORWARD**" and "**40**," you'll see the message "**I DON'T KNOW HOW TO FORWARD40**." You'll get a similar message if you make any other typing error. Just press $\boxed{\text{ENTER}}$ to get another chance to enter "**FORWARD 40**." You can use the left-arrow key to correct typing errors before you press $\boxed{\text{ENTER}}$. Simply backspace to the beginning of the error, and retype the turtle instruction.

Now I want you to find out how far it is from the center of the screen to the top. To get a fresh start and a clear screen, enter the word (not the single key)

```
CLEAR
```

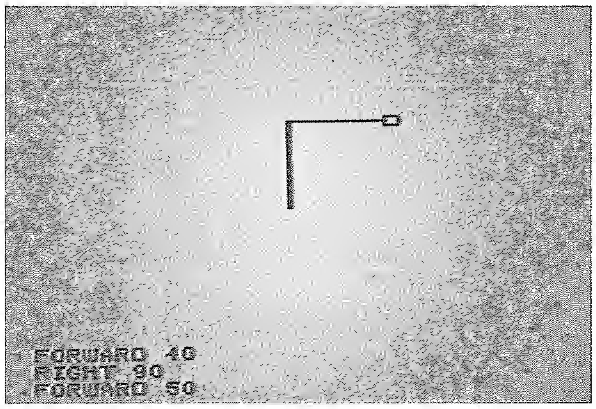
Then try to make the turtle track go to the top of the screen with a single **FD** command. When you have it exactly right the turtle itself will wrap around (appearing at the bottom of the screen), but the line will be drawn to the top of the screen. No doubt it will take you several tries of **CLEAR**, **FD** to hit the top exactly.

By now you're probably tired of drawing vertical lines. It's time to turn the turtle. Clear the screen (**CLEAR** ENTER) and enter

```
FORWARD 4Ø
RIGHT 9Ø
```

To make the change more obvious enter

```
FORWARD 5Ø
```



```
FORWARD 40
RIGHT 90
FORWARD 50
```

The turtle understands degrees.

If you are using Color LOGO with small children, we have a suggestion. There is now quite a bit of information gathered about effective use of LOGO with small children, and that information applies to the use of Color LOGO as well. Color LOGO is a language for experimentation, not a language to learn by imitation of items from a textbook. Resist any temptation to explain degrees to the child who does not already know about them. The child will learn about degrees easily from experimenting with Color LOGO.

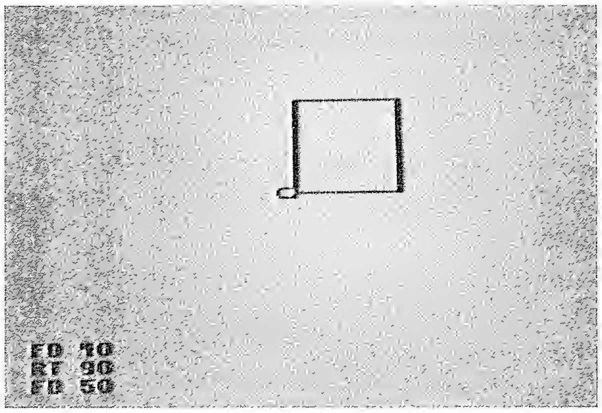
Again we soon get tired of typing **RIGHT**, so we abbreviate **RT**. Try

```
RT 9Ø
```

(Think "right turn" for **RT**.) Now the turtle points down. We're half way to drawing a rectangle, so let's finish it. Enter

```
FD 40
RT 90
```

and see if you can finish it.



```
FD 40
RT 90
FD 50
```

Once you've finished the rectangle (by entering **FD 50**), clear the screen and enter

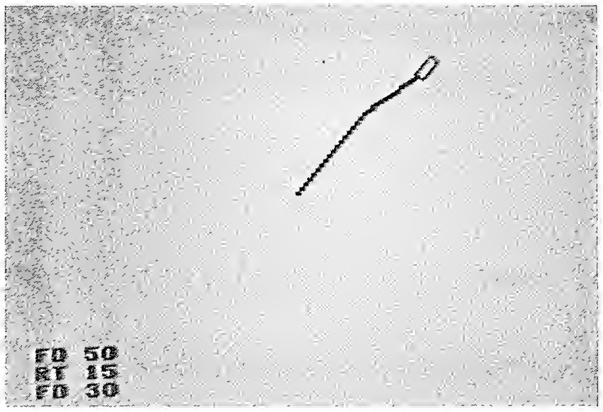
```
RT 45
FD 50
```

Just what you expected, I hope, but now try

```
RT 15
```

It looks like nothing happened! But to check try

```
FD 30
```



```
FD 50
RT 15
FD 30
```

Obviously there is a bend in the line even though the turtle did not appear to turn. The turtle knows its heading to the nearest degree and moves accordingly, but the turtle shape on the screen turns only in 45 degree steps. Thus the turtle shape on the screen points close to, but not exactly along, the turtle heading. This seems inconvenient now, but in Chapter 14 we'll find that there are some real benefits of this.

We want you to notice one very important thing about turtle behavior. Clear the screen and enter

```
RT 45
RT 45
```

This produces the same heading as **RT 90**. When the turtle is told to turn, it turns that far from whatever its current heading is. We are telling the turtle how to change its heading; we are *not* telling the turtle to head towards some point. In the same way, when we tell the turtle to go forward we are telling the turtle how to change its position; we are not telling the turtle to go to some point on the screen. Thus the position and heading of the turtle after one of these commands will depend on where the turtle started.

Thus far we have learned three primitive turtle commands. (Papert would say, three words in "turtle talk.") They are **CLEAR**, **FORWARD**, and **RIGHT**. With these three we can draw any figure which will fit on the screen and which could be drawn on paper without lifting the pencil from the paper. You might try drawing a triangle (3-sided figure) and a pentagon (5-sided figure) for practice. If you're like us, you don't remember the angles for pentagons, so experiment.

You've probably noticed that only three lines of instructions are shown at any one time. When you type a fourth line the lines scroll up, and the top line disappears. If you have drawn a turtle track through these three bottom lines, the scrolling will mess up the line by scrolling a line segment upwards. If you leave the turtle in these three bottom lines while scrolling, the turtle will not erase properly when moving to another spot. These are minor consequences of some characteristics of the Color Computer which will not trouble us once we get to Chapter 4. For now, either avoid moving the turtle into the bottom three lines, or press ENTER enough times to scroll away the line and turtle fragments, or ignore the fragments.

We could go a long, long way with just **RIGHT** and **FORWARD**, but **LEFT** and **BACK** are useful too. Clear the screen and try

```
LEFT 90
```

(We could have used the abbreviation **LT** for "left turn.") Now let's make the turtle move backwards. Try

```
BACK 40
```

(or in abbreviated form **BK 40**). Notice that the turtle is somewhat transparent. You can see the track through the turtle. If you'd rather not see the turtle at all, you can hide it. Enter

```
HIDETURTLE
```

(Here the abbreviation **HT** is much shorter.) The turtle is still there but invisible. Enter

```
LT 30
BK 30
```

to see the invisible turtle's track. To make the turtle visible again type

```
SHOWTURTLE
```

(you guessed it, abbreviated **ST**), and to turn it away from the track type

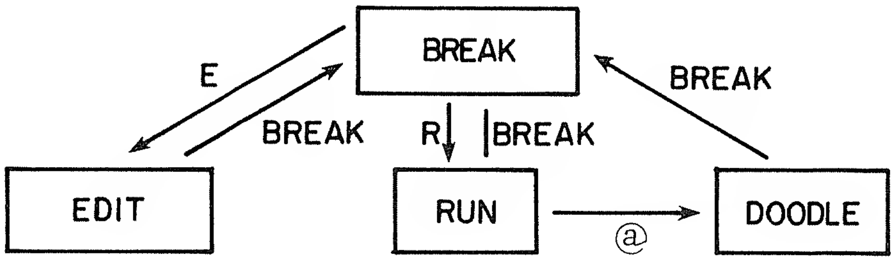
```
LT 120
```

At this point the only thing between us and an endless variety of stunning graphics is an immense amount of typing. To learn how to minimize the typing we have to learn how to create and use procedures.

3. MODES AND EDITING

New users of computers often find the idea of modes awkward. Mode is the term used to describe the separation of the various things a computer language can do into groups. There are a number of good reasons for having various modes. One is that there are not enough different keys on the keyboard to control all the different things that need to be done. The same keys can be used for different tasks in different modes without confusion (at least on the computer's part).

The following diagram is a map of the modes in Color LOGO.



The keys which trigger the jumps between modes are indicated on the arrows. You've already been in BREAK mode; that's the mode that you are in when you start. You've already been in RUN mode; you got there from BREAK mode by pressing R. Now we want to move into EDIT mode. The map shows us that we need to leave RUN mode (by pressing the BREAK key) and then get into EDIT mode (by pressing the E key).

EDIT mode provides what is called a line-oriented editor. EDIT mode is used to create and alter programs written in Color LOGO, but for the rest of this chapter we will forget Color LOGO and concentrate on the mechanics of using EDIT mode. We'll do something familiar—write a note to Grandma.

Upon entering EDIT mode, you'll see a short horizontal line appear at the start of the bottom line of the screen. This line is called the cursor. The cursor indicates where any typed letters, numbers, etc., will appear. Start the note by typing

DEAR GRANDMA,

Press ENTER, and the cursor moves to the start of the next line. Type the next line as

I'M STARTING TO USE AN EDITOR.

Again press ENTER to complete the line. Notice that this editor produces only upper-case letters; Color LOGO uses only upper-case letters.

We could continue to enter as many lines as we wanted in the same fashion. Let's assume that this is to be a very short note and that we now want to quit editing. Press BREAK. Upon reflection we decide to alter the note, so we return to EDIT mode (press E). The first line of our note appears with the cursor at the start of the line. We decide to change the word "**STARTING**" in the second line of the note to the word "**BEGINNING**." To do this, we must first display the second line and position the cursor under the **S** in **STARTING**. We move the cursor by use of the arrow keys. Up-arrow (↑) and down-arrow (↓) change lines, and left-arrow (←) and right-arrow (→) move the cursor within a line. Changing lines always resets the cursor to the start of the line. Arrow commands which make no sense are ignored. Thus if we press right-arrow when the cursor is under the comma following **GRANDMA**, nothing happens because there is no more line.

To see the second line of the note, press the up-arrow key once. Then press the right-arrow key several times to position the cursor under the **S** in **STARTING**. Then type

```
BEGIN
```

Notice that the overtyping simply replaces the letters. Now we have another kind of change to make because **BEGINNING** has one more letter than **STARTING**. We want space for another **N** before the **ING**. To create a space we hold down the SHIFT key and press the right-arrow key. Now we can type the extra **N** in the created space. Remember: to insert, press SHIFT → to create the space, then type in what you want.

Next let's change the line from

```
I'M BEGINNING TO USE AN EDITOR.
```

to

```
I'M LEARNING TO USE AN EDITOR.
```

Again position the cursor at the start of **BEGINNING** and type over the characters you want to change. Here the problem is that an extra **N** remains. To delete a character (or space) hold down the SHIFT key and press the left-arrow key. Try it, and remember: press SHIFT ← to delete.

Poor Grandma isn't going to know who the note is from unless we add a line at the end. Use the up-arrow key to move the cursor as far down as you can. It should be at the start of a blank line following the text. We want to skip a line before signing the note, so press ENTER once. Notice that pressing ENTER adds a line at the end. But if the cursor is within the text, pressing ENTER has the same effect as the up-arrow. Now space over and sign your name. While we are at it, we should skip a line after **DEAR GRANDMA**. That is, we want to change

```
DEAR GRANDMA,
  I'M LEARNING TO USE AN EDITOR.

                    LOVE,  ANN
```

to

```
DEAR GRANDMA,

  I'M LEARNING TO USE AN EDITOR.

                    LOVE,  ANN
```

Position the cursor at the beginning of the line "I'M...". Then hold down the SHIFT key and press the down-arrow key. Move the cursor down to check that you got what you wanted. Remember: to insert a new line, position the cursor at the start of the following line; then press SHIFT ↓ .

We want to make one last change. We want to change the closing to

```
LOVE,
    ANN
```

We want to break one line into two. Position the cursor where you want the break to occur; then press SHIFT ↓ to break the line. You'll have to insert some spaces to move the name over as shown above.

This we think is the final form of the note, so we exit EDIT mode (press BREAK). To make a last check we get back into EDIT mode (press E). To get the whole note on the screen without repeated pressing of up-arrow or ENTER we press SHIFT ↑. This will show us everything in memory. If we want to interrupt this process, just press any key to stop the scan. To restart the scan, press SHIFT ↑ again. To jump back to the start of the text, press CLEAR .

That is all there is to using the editor. We suggest that you practice a bit with it so that when we return to Color LOGO you can concentrate on the language and not have to worry about the mechanics of the editor.

To conclude this chapter we give a summary of the editing features.

To:	**Press:**
get into EDIT mode	BREAK, E
display the next line of text	↑ or ENTER (↑ has no effect at last line) (ENTER adds line after last line)
add a line at end	ENTER, type line of text
move text down one line	↓ (no effect at top line)
move cursor right	→ (no effect at line end)
move cursor left	← (no effect at line start)
replace character	position cursor, type over error
insert character	position cursor, SHIFT → (no effect if line full), type character
delete character	position cursor, SHIFT ←

insert line	position cursor at start of following line, $\boxed{\text{SHIFT}}\boxed{\downarrow}$
break line	position cursor at break point, $\boxed{\text{SHIFT}}\boxed{\downarrow}$
return to top line	$\boxed{\text{CLEAR}}$
scroll or scan through text	$\boxed{\text{SHIFT}}\boxed{\uparrow}$
stop scroll or scan	press any key.

4. PROCEDURES

You've now mastered five primitive turtle commands (**CLEAR**, **FORWARD**, **BACK**, **RIGHT**, and **LEFT**). Next we want to combine these commands into a unit which we call a procedure. The first step is to tell the computer not to obey each command as it is typed, but to store the commands. This is what happens in EDIT mode. Press **BREAK**, then hold **SHIFT** down and press **CLEAR** (to clear the memory of old programs). Then get into EDIT mode (press **E**).

The screen should be blank with the cursor in the lower left corner. If the screen is not blank, return to **BREAK** mode (by pressing the **BREAK** key), hold the **SHIFT** key down, and press the **CLEAR** key firmly. Return to EDIT mode by pressing **E**.

You are now using a line-oriented editor. We will practice using the editor as we create and edit procedures. Our first exercise will be to write a procedure for drawing a rectangle. First we must give the procedure a name. We'll call this first one "**RECTANGLE**." The first line of the procedure contains the name, and we let the computer know that we're naming a procedure by starting the first line with the keyword "**TO**." To name this first procedure **RECTANGLE**, enter

```
TO RECTANGLE
```

The limitations on procedure names are that they must fit on a single line, they must contain no spaces, and they must not be the same as any of the keywords or abbreviations (for example, **FORWARD** or **FD**).

If you made a typing error when you were using RUN mode, you got the error message "**I DON'T KNOW HOW TO**" followed by your mistyped command. Because a procedure name can be almost anything, the computer assumes that any characters which don't form a correct keyword must form a procedure name. If the characters are really a typing error, then the name is not found in the list of procedures and the error message is sent.

Next type in the turtle commands for drawing the rectangle. That is, type

```
FD 50 RT 90 FD 30 RT 90 FD 50
RT 90 FD 30
```

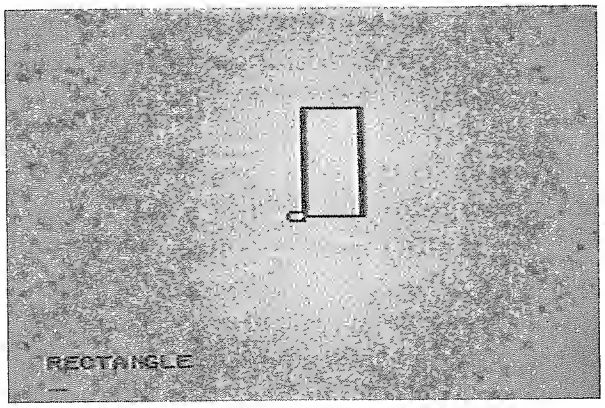
Many commands can be typed on a single line as long as they are separated by one or more spaces. To finish the procedure type

```
END
```

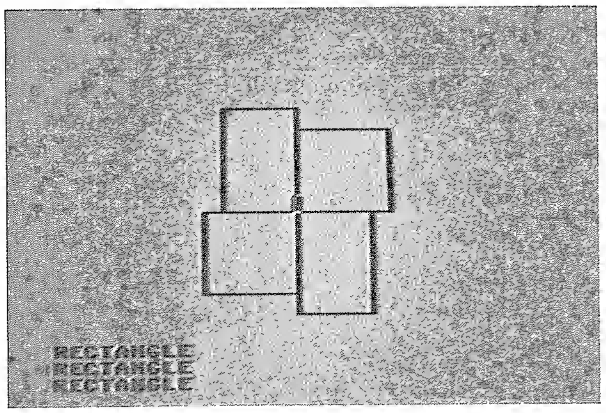
on a new line and press **ENTER**.

To try out **RECTANGLE**, you must leave EDIT mode (by pressing the BREAK key) and get into the RUN mode (by pressing the R key). To actually run the procedure, enter

```
RECTANGLE
```


RECTANGLE

That's so neat that we should try it again and again. Type and enter the procedure name at least three more times. Now the screen should show


RECTANGLE
RECTANGLE
RECTANGLE

By placing procedure **RECTANGLE** in the computer's memory we have taught the turtle to understand a new word. The turtle now understands **RECTANGLE** in the same way that it understands **LEFT**, **RIGHT**, **FORWARD**, and **BACK**.

Before moving on to other procedures, we want to review use of the editor. Press BREAK to return to BREAK mode and press E to reenter EDIT mode. The screen should now show the first line of the procedure **RECTANGLE**. Let's change the name to "**BOX**." Use the right-arrow key (→) to position the cursor under the R in **RECTANGLE**. Then type **BOX**. Remember, overtyping replaces characters. We need to delete the remaining letters, which we do by holding down the SHIFT key and pressing the left-arrow key (←). We can see the rest of the lines in the procedure by pressing either ENTER or the up-arrow key (↑) several times.

It is good programming practice to clarify the structure of a procedure by indentation. Here we want the procedure BOX to look like this

```
TO BOX
  FD 50 RT 90 FD 30 RT 90 FD 50
  RT 90 FD 30
END
```

To make these changes we must insert a couple spaces at the beginnings of the second and third lines. Move the second line to the bottom of the screen by using the up- and down-arrow keys. The cursor will move to the start of the line whenever you change lines. To insert spaces, hold down the SHIFT key and press the right-arrow key. If this does not insert spaces, it means that the line is already full. Insert spaces at the start of line 3 as well.

The structure of the procedure would be even clearer if it were typed as follows.

```
TO BOX
  FD 50   RT 90
  FD 30   RT 90
  FD 50   RT 90
  FD 30
END
```

These changes require us to break single lines into multiple lines. To break a line, position the cursor where you want to break the line, hold the SHIFT key down and press the down-arrow key.

What if we want to add lines to a procedure; for example if we want to add a diagonal line through the box? We'll have to tell the turtle to turn and go forward. You'd better run BOX to get an estimate of the angle and distance (remember press BREAK, then press R, then enter **BOX**). The turtle needs to be turned more than 90 degrees to point along the diagonal. Make a guess and return to EDIT mode (BREAK, E). Now place the cursor under the **E** in **END**; hold down SHIFT and press ↓. This inserts a blank line (try the up-arrow key to check that **END** has just been bumped down one line). You can now insert your **RT** and **FD** commands in this new blank line. It will no doubt take you several tries to get the angle and length exactly right; that will give you good practice in bouncing back and forth between RUN and EDIT modes. (No fair using your knowledge of trigonometry; with turtles you are supposed to experiment.)

In this chapter we have covered two main topics. We have learned how to enter and change multiple-command procedures, and we have learned how to teach the turtle to understand more complex commands via procedures.

5. REPEAT AND SUBPROCEDURES

Once we have taught the turtle a new word by writing a procedure, we can use that new word in further procedures. Return to EDIT mode and remove the commands for drawing the diagonal (I used **RT 122 FD 59**) from **BOX**. Now move to a new line (press **ENTER**). In fact a blank line between procedures will help keep things easy to read, so press **ENTER** again. We're going to write another procedure to draw the pattern of four boxes. We'll call it "**FOUR**," so type

```
TO FOUR
  BOX
  BOX
  BOX
  BOX
END
```

Notice that we've used **BOX** as a turtle command in the same way that we used **FORWARD** and **RIGHT** within **BOX**. Run **FOUR** to see that it works. The result is the same as that shown on page 18.

To run the procedure **FOUR**, the computer must have available the subprocedure **BOX**. Both procedures must be in the program space when **FOUR** is run, but their order within that space is of no importance. We could have written **FOUR** first and then written **BOX** with exactly the same result.

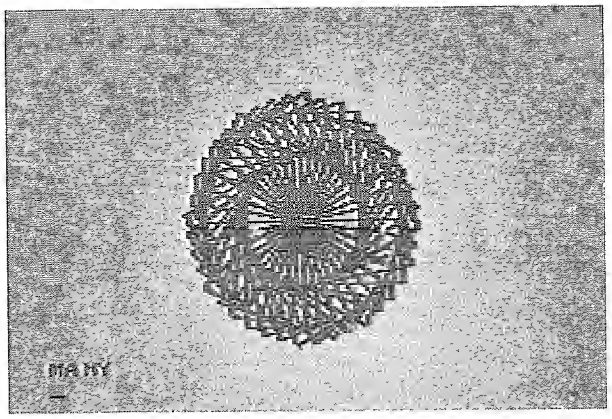
The procedure **FOUR** can be shortened by use of the **REPEAT** control statement. The altered form of **FOUR** is

```
TO FOUR
  REPEAT 4 (BOX)
END
```

The **REPEAT** tells the turtle to repeat the actions within the parentheses the designated number of times, in this case, four. The space after the number 4 is optional. The parentheses can include a whole list of turtle commands and subprocedure names. The list in parentheses can extend over many lines, but the parentheses are essential.

Now that we have taught the turtle what **FOUR** means, we can move to a higher-level procedure. Try

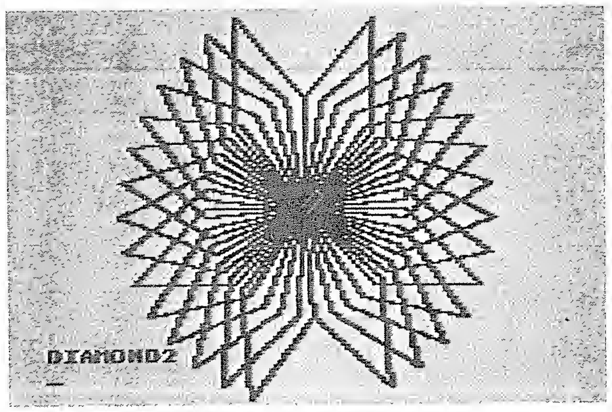
```
TO MANY
  REPEAT 10 (FOUR   RT 9)
END
```



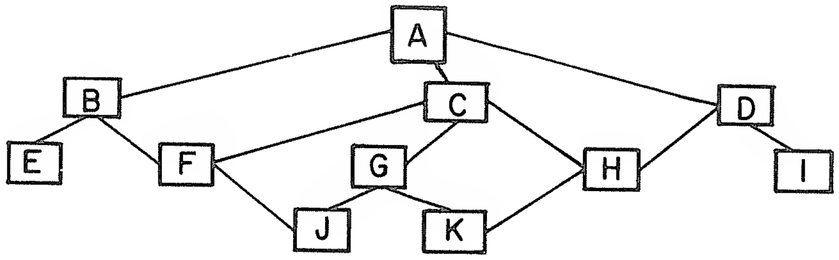
By now you probably are tired of following the manual and are consumed with curiosity. What would happen if I changed the number on the **REPEAT** in **MANY**; what would happen if I changed the angle in **MANY**; what would happen if I restored the commands to draw the diagonal in **BOX**? Don't hesitate to find out by trying; that's the whole point of Color LOGO. Try triangles, pentagons, hexagons, threes and fives instead of just boxes and fours.

Here is another sample.

```
TO DIAMOND
  FD 50  LT 45  FD 50  LT 135
  FD 50  LT 45  FD 50
END

TO DIAMOND2
  REPEAT 29  (DIAMOND   RT 40)
END
```



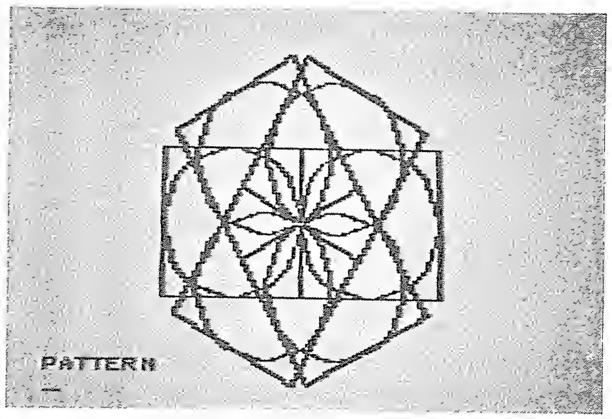
Color LOGO is a structured language. A complex program written in Color LOGO could have the following structure.



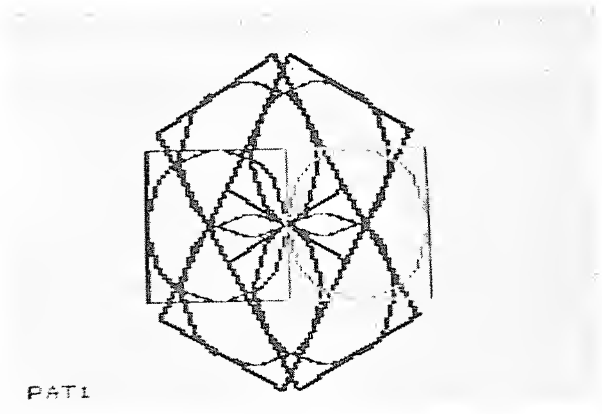
Each letter within a box represents a procedure; each line of type on the page includes the subprocedures of a particular level; the lines indicate which subprocedures are used by each procedure. There are four levels of procedures within this program. The master procedure A (level Ø) might use the subprocedures of level 1 in the order B, C, D, C. Subprocedure B might use the subprocedures of level 2 in the order E, F, E; subprocedure C might use the subprocedures of level 2 in the order G, F, H, etc. Notice that subprocedures can be used many times and many places within the overall program.

Thus far in our examples we have been working from the bottom up, defining a first procedure, then writing a second procedure that uses the first procedure as a subprocedure, etc. That is typical of programming manuals where the emphasis is on the mechanics of a language instead of on problem solving. It is especially fun to adopt that approach with Color LOGO at times because the results are often unpredictable and interesting. However, as we become more serious we often will have a problem we wish to solve. Then we should work from the top level down. Now we illustrate that process.

The sample problem is to create the following pattern.



Obviously the figure is so symmetrical that there is a repeat pattern. Furthermore the number of repeats must be six. The crucial step is to recognize that the element that is repeated six times is a square with a circle inside.



Therefore our main procedure should be

```
TO PATTERN
   REPEAT 6 (SQUARE-CIRCLE   RT 60)
END
```

The six-fold symmetry tells us to repeat 6 times with turns of 60 (as $6 * 60 = 360$). As yet we have no idea how to draw a square with a circle inside.

Now we move to the next lower level.

```
TO SQUARE-CIRCLE
   SQUARE
   CIRCLE
END
```

Again we break the task into simpler tasks. This time the breakdown is obvious; you draw a square with a circle inside by drawing a square and then a circle.

Now we drop down to level 2. The obvious procedure for drawing a square is

```
TO SQUARE
   REPEAT 4  (FD 70   RT 90)
END
```

This will draw a square, but it will leave us with a problem. We also have to draw a circle inside the square, and **SQUARE** leaves the turtle at a corner of the square. The corner is an awkward place to start drawing a circle which is to be inside the square. This example shows that when procedures are to be used together some attention must be devoted to making them fit. The circle and the square touch at the center of a side. We choose to make the two procedures fit by starting and ending the square at the center of a side.

```
TO SQUARE
  REPEAT 4 (FD 35  RT 90  FD 35)
END
```

Now how do we get the turtle to draw a circle? One very good way to figure this out is to play turtle. That is, walk in a circle and think about what you are doing in terms of turtle commands. You'll discover that to make a circle you go forward a little and turn a little until you get all the way around. The best circle should be drawn by the following, right?

```
TO CIRCLE
  REPEAT 360 (FD 1  RT 1)
END
```

Wrong! This gives an eight-sided figure. (I'll explain why, but if you find this confusing for now, skip the next paragraph.) When a turtle moves forward one step it has eight choices of where to go.

8	1	2
7	X	3
6	5	4

We start with the turtle pointing straight up (towards 1) and turn the turtle a small amount, say 5 degrees right. Clearly the only two choices are to go to square 1 or to square 2, and square 1 is closer to the correct direction than square 2. Because Color LOGO does only integer arithmetic the turtle is now assumed to be in the center of square 1 and the process repeats on the next move. With larger steps there are more squares to which the turtle can move, and we get a better circle. In general, avoid combining tiny steps with tiny turns for turtles.

A reasonable circle can be drawn by the procedure

```
TO CIRCLE
  REPEAT 36 (FD 6  RT 10)
END
```

This gives the same problem as the first procedure for the square: the circle will not be centered in the square. We can fix the procedure if we realize that what we are really drawing is a 36-sided polygon which looks like a circle because of the limited screen resolution. The fix is the same as before; start in the center of the side.

```
TO CIRCLE
  REPEAT 36 (FD 3  RT 10  FD 3)
END
```

Now run **PATTERN** to verify that this set of procedures is a solution to the original problem.

Let's analyze what we've just done. The road map for attacking the problem was to break the problem into a set of subproblems, and in turn to break each subproblem into even simpler subproblems until the subproblems can be solved by a single **REPEAT** statement. Specifically, we broke the original problem into the problem of drawing **SQUARE-CIRCLE** six times; we broke **SQUARE-CIRCLE** into the problems of drawing a square and drawing a circle. These last two problems were easily solved with a single **REPEAT** statement. In general, we follow this sequence in attacking a problem, although we do not insist that the lowest level procedure consist of a single **REPEAT** statement.

One of the reasons for using Color LOGO with children is that it is an excellent way to teach children a most powerful and useful general problem-solving approach. That approach is what we have just illustrated. Basically, it involves working from the overall view down to the details by breaking each problem into pieces.

6. VARIABLES

Variable is the name used to describe unique storage locations where numbers can be kept. The name of a variable consists of a colon (:) followed by any number of letters and/or numbers. Variables can be used anywhere numbers can be used; in this way procedures can be used for a much wider range of applications.

A typical use of a variable is the following:

```
TO SQUARE  :SIDE
  REPEAT 4  (FD :SIDE  RT 90)
END
```

If you came here directly from the last chapter, then there is another version of **SQUARE** in memory. To clear out the memory, press $\boxed{\text{SHIFT}}$ $\boxed{\text{CLEAR}}$ while you are in BREAK mode. Then get into the EDIT mode and enter the new version of **SQUARE**. Now to run **SQUARE** get into RUN mode and enter

```
SQUARE 40
```

Because we have listed the variable **:SIDE** on the **TO** statement, we must give a value when we call (or use) the procedure **SQUARE**. Now try a variety of other numbers, for instance

```
SQUARE 60
SQUARE 20
```

What happens if we forget the number? Try

```
SQUARE
```

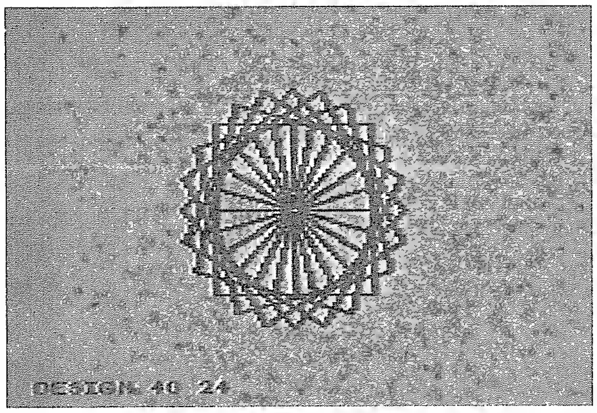
If we don't provide a number, then the computer provides a zero. The brief flicker is due to the turtle turning in place while drawing a square with zero-length sides.

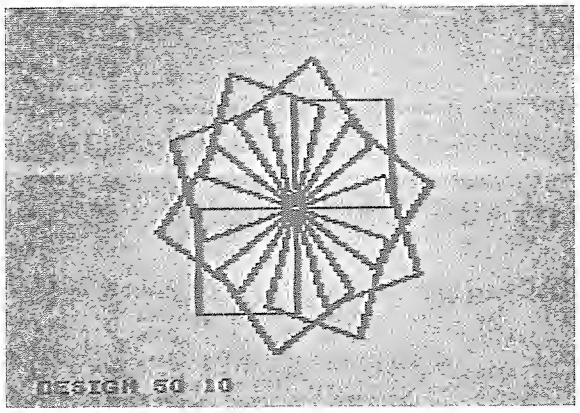
Variables can be used in other positions as well. Here's another example.

```
TO DESIGN  :LENGTH  :TIMES
  REPEAT  :TIMES  (SQUARE  :LENGTH
                  RT 360/:TIMES)
END
```

Enter this and try running it with a few different values of **:TIMES** and **:LENGTH**. For example:

```
DESIGN 40 24
DESIGN 50 10
```





The computer keeps track of the variables by the order. Because the order in the **TO** statement for **DESIGN** is **:LENGTH :TIMES**, the command **DESIGN 40 24** causes the value **40** to be assigned to **:LENGTH** and the value **24** to be assigned to **:TIMES**. Notice also that the name of the variable in the call of **SQUARE (SQUARE :LENGTH)** need not be the same as the name in the definition of **SQUARE (TO SQUARE :SIDE)**. At the time **SQUARE** is called **:LENGTH** has a value (for example, **40**) which is assigned by **SQUARE** to the variable **:SIDE**.

Variables listed on the **TO** statement are local to the procedure. Again we illustrate using the previous programs. Enter:

```
TO DESIGN  :LENGTH  :N
   REPEAT  :N  (SQUARE  :LENGTH
                         RT 360/:N)
END

TO SQUARE  :N
   REPEAT 4  (FD  :N   RT 90)
END
```

Here the variable name **:N** is used for two different quantities, one in the main procedure **DESIGN** and another in the subprocedure **SQUARE**. This causes no problems or confusion because the variables for the two procedures are kept completely separate in the memory. The variable **:N** in the main procedure is not the same memory location as the variable **:N** in the subprocedure.

If we want a variable to be local to a procedure, we mention it in the **TO** statement which begins the procedure. The maximum number of local variables for a procedure is five. We also can create global variables, variables which use a common memory location in all procedures in which they appear. Global variables are created by using them in a procedure without including them in the TO statement. This provides a convenient way to share information among procedures.

DESIGN contains our first example of arithmetic expressions, here **360/:N**. Color LOGO allows the standard four arithmetic operations: addition (+), subtraction (−), multiplication (*), and division (/). No parentheses are needed unless the order of operations is non-standard. Fractional parts of numbers are dropped. Thus in Color LOGO

```
5 / 3 = 1
2 * 3 + 4 = 10
2 * (3 + 4) = 14
```

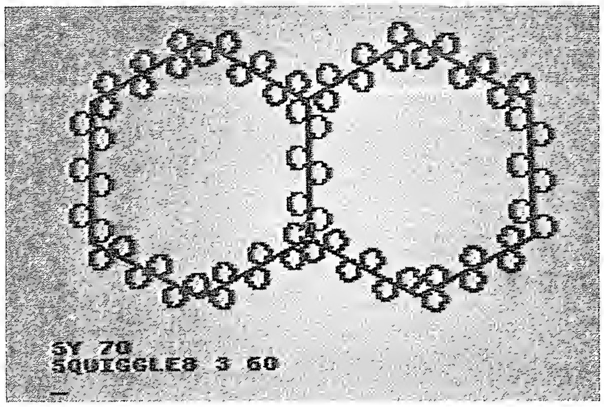
Numbers in the range -32768 to 32767 can be handled. To throw away the sign of a number, use the **ABS** function ("**ABS -10**" is "+ 10").

The following procedures give additional examples of the use of variables and arithmetic expressions.

```
TO SQUIGGLE
  FD 7
  REPEAT 8
      (FD 4   RT 45)
  FD 7
  REPEAT 8
      (FD 4   LT 45)
  FD 7
END

TO SQUIGGLE8 :SIDE :ANGLE
  REPEAT 360/:ANGLE
      (REPEAT :SIDE (SQUIGGLE)
          RT :ANGLE)
  REPEAT 360/:ANGLE
      (REPEAT :SIDE (SQUIGGLE)
          LT :ANGLE)
END
```

Notice the use of the nested **REPEAT** twice in **SQUIGGLE8**. If **:ANGLE** were **180** and **:SIDE** were **4**, then each of the pairs of nested **REPEAT**s will repeat 2*4 or 8 times. Try

```
SQUIGGLE8  1  20
SQUIGGLE8  3  60
SQUIGGLE8  4  90
```



```
SY 70
SQUIGGLES 3 60
```

7. COLORS

Turtle tracks can be colored, and they can change color. Your TRS-80 Color Computer offers two color "sets" (or "settings") in the high resolution screen on which turtles live. Thus far you have been running in color set Ø. You can shift color sets by the **COLORSET** command. Get into the RUN mode and enter

```
COLORSET 1
```

to change color set. Then enter

```
COLORSET Ø
```

to change back.

Within each color set there are four colors, numbered Ø, 1, 2, and 3. The normal drawing color is color Ø and the normal background color is color 3. Change the background color by entering

```
BACKGROUND 1
```

or abbreviate

```
BG 1
```

Change the pen (or drawing) color by entering

```
PENCOLOR 2
```

or abbreviate

```
PC 2
```

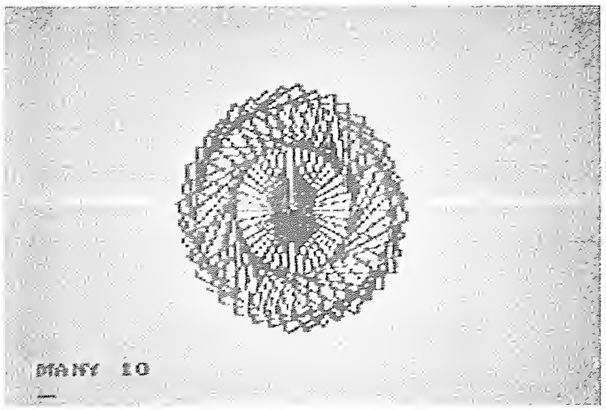
A portion of a drawing can be erased by setting the pen color to the same color as the background color is set and drawing over the unwanted part of the drawing.

Let's add color to some of our earlier procedures. One interesting choice is **FOUR**. Retype **BOX** (see page 19), then enter the procedure **FOUR** as:

```
TO FOUR
  REPEAT 2  (PC 1  BOX
             PC 2  BOX)
END
```

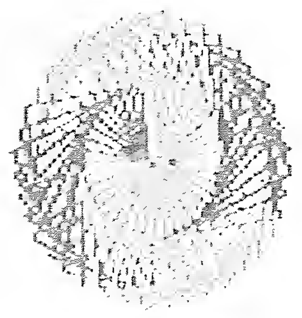
and to allow easy experimentation, make **MANY** into

```
TO MANY  :N
  REPEAT :N  (FOUR   RT 9Ø/:N)
END
```


MANY 10

We would like to be able to name the colors you will get with specific pen colors and color sets, but colors vary from TV to TV, they vary with the color settings on the TV, and they may even switch when you restart your computer. Try running **MANY** with a value of 1Ø. Then adjust the color and tint controls on your TV set to your satisfaction. On many TV's, color set 1 will give more interesting colors, so be sure to try that too. You can change color sets without redrawing the figure by typing

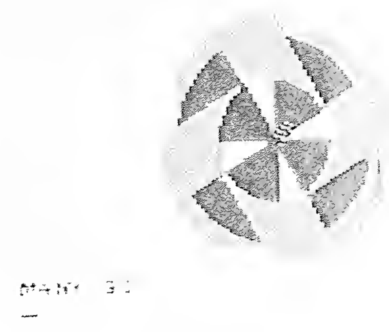
```
COLORSET 1
```

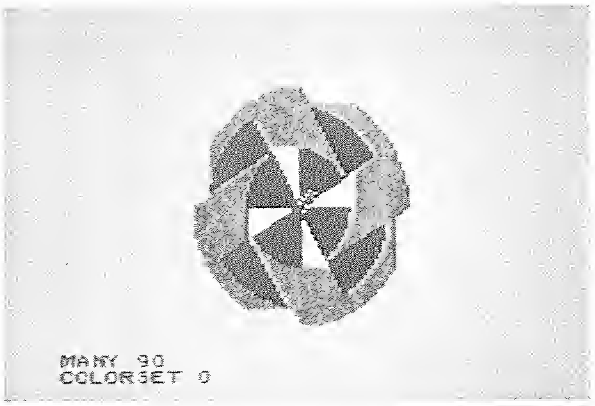

```
MANY 10
COLORSET 1
```

An interesting variation can be created by the following changes.

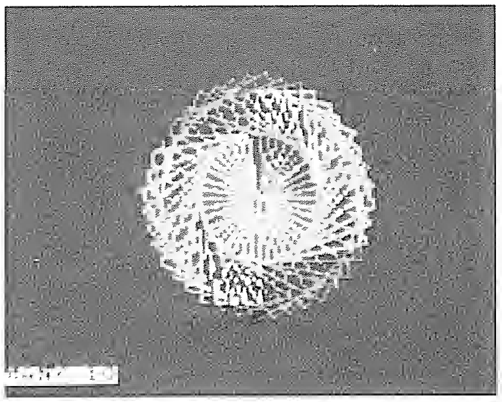
```
TO BOX
  PC 1  FD 5Ø  RT 9Ø
  PC 2  FD 3Ø  RT 9Ø
        FD 5Ø  RT 9Ø
  PC 1  FD 3Ø
END

TO FOUR
  REPEAT 4  (BOX)
END

TO MANY  :N
  REPEAT 2*:N/3  (FOUR  RT 9Ø/:N)
END
```

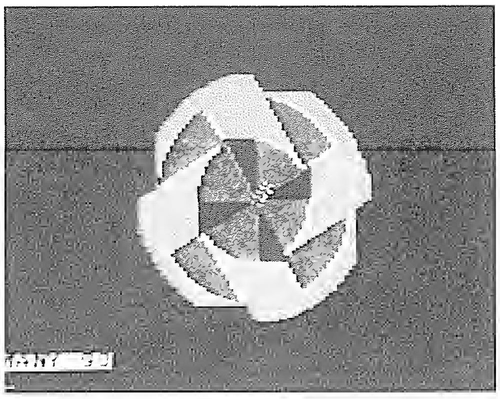
Try it with :**N** = **90**. If it is too slow, or if you feel sorry for any turtle that has to run around at that speed for so long, hide the turtle (**HT**) before calling **MANY**.





You might prefer the colors you get with a dark background. Try setting the background to 0, and rerun the preceding two examples, **MANY 10** and **MANY 90**.





8. OTHER TURTLE COMMANDS

There are a few additional turtle commands which we have not yet used. We can raise and lower the turtle's tail, so we have the choice of leaving a track or not leaving a track. The commands are just what you'd guess, **PENUP** (abbreviated **PU**) and **PENDOWN** (abbreviated **PD**).

Let's illustrate by removing the lines of one color from the previous figure. Change **BOX** to

```
TO BOX
   PU  FD 50  RT 90
   PD  PC 2   FD 30  RT 90
              FD 50  RT 90
   PU  FD 30
END
```

and again run **MANY 90**.

Every figure thus far has started in the center of the screen at a position called home. When we get into the RUN mode the turtle automatically moves to home. If we want to start the turtle somewhere else, we can. The turtle can be moved to an arbitrary and absolute screen position by means of the **SETX** (abbreviated **SX**) and the **SETY** (abbreviated **SY**) commands. The results of these two commands are absolute, not relative to the current position of the turtle. No line is drawn, and no change in heading is made. For example,

```
TO DOUBLE
   SETX  60
   MANY 90
   SETX  180
   MANY 90
END
```

The heading of the turtle also can be set to an absolute value independent of its current heading. The command is **SETHEADING** (abbreviated **SH** or **SETH**). The heading can be anything between Ø and 359 degrees. Zero degrees is straight up. Try

```
TO DOUBLE
  HT
  SX 6Ø
  MANY 9Ø
  SX 18Ø  SH Ø
  MANY 9Ø
END
```

The remaining turtle instruction is **HOME**. **HOME** returns the turtle to the home position (the center of the screen) with a heading of Ø degrees (straight up).

Procedures which draw circles and parts of circles (arcs) are very useful in other projects. There are some drawbacks to the **CIRCLE** procedure given in Chapter 5, page 25. It's hard to predict the size of the circle from the size of the step, it's hard to find the optimum number of steps for the best circle, and it's hard to figure where the circle is centered. The following procedures are a useful alternate set.

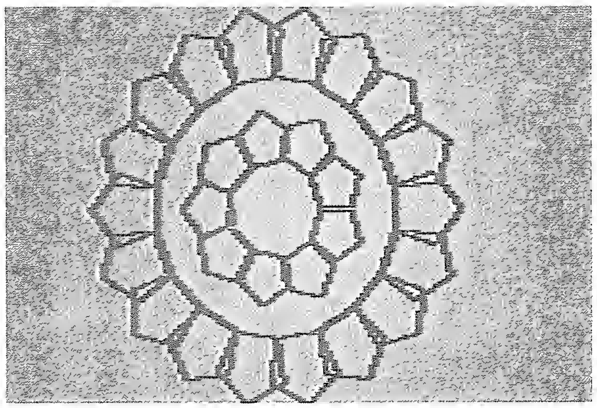
```
TO ARC  :X  :Y  :RADIUS  :DEGREE
  PU  SX  :X  SY  :Y
  REPEAT  :DEGREE  (FD  :RADIUS
       DOT  BK  :RADIUS   RT 1)
END

TO DOT
  PD  FD 1  BK 1  PU
END
```

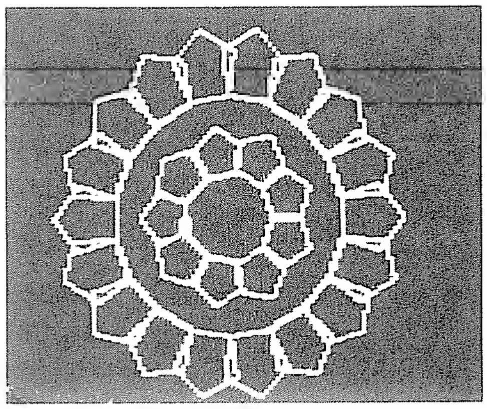
These are slower than **CIRCLE**, but they give the most accurate circles which can be drawn on the screen. With some values of **:RADIUS** you might be able to get the same accuracy with fewer steps (for example, **REPEAT 18Ø** and **RT 2**), but then the number of degrees will have to be divided to get the right number on the **REPEAT**. You also might want to enlarge the dots.

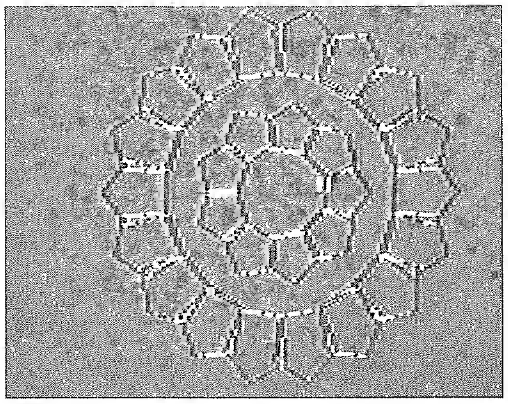
```
TO DOT
  RT 9Ø  BK 1  PD  FD 1  FD 1  BK 1
  PU  LT 9Ø
END
```

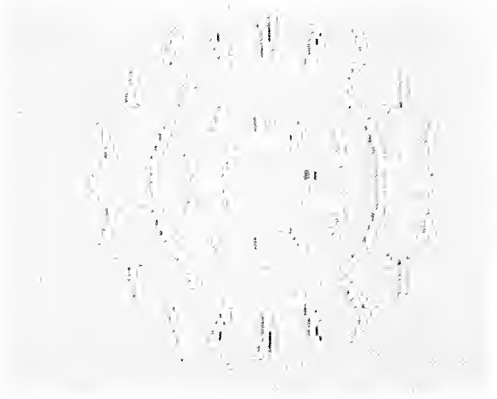
The following program again makes use of the **SX** and **SY**, here to get the right relative spacing of independent parts. The correct numbers for the two are found by trial and error. Try the procedures with a variety of pen colors and backgrounds:

```
TO KIRSTIN
   CLEAR   SX 60  SY 80
   REPEAT 18  (PENT 20  RT 20)
   SX 95  SY 82
   REPEAT 9   (PENT 15  RT 40)
END

TO PENT  :SIDE
   REPEAT 5  (FD  :SIDE  LT 72)
   FD  :SIDE
END
```









9. SAVING, LOADING, AND PRINTING YOUR COLOR LOGO PROCEDURES

Although Color LOGO procedures can do an amazing amount with very little code, we still don't want to have to retype procedures every time we start up. Procedures can be stored on cassette tape or on disk (just cassette on ROM version). For simplicity all procedures in memory are stored as one module, and all procedures in a module on disk or tape are loaded. No directories are kept for the modules (a directory entry could be almost as long as many of the procedures in Color LOGO).

To move procedures to and from cassette or diskette we must be in BREAK mode. By now you probably have a number of procedures in memory that you would like to keep. Get into EDIT mode and delete any you do not want to save. (To delete a procedure you must delete it character by character.) Then get into BREAK mode (**BREAK** key) and press **S**. At this point the prompt will be

```
LOGO: SAVE:_
```

You now have to tell the computer where to save the procedures in memory. Let's first learn how to save on tape. Of course the tape recorder must be plugged in as described in the Operation Manual for the TRS-80® Color Computer. Make sure that the volume control is set close to 5. Rewind the tape (REWIND, STOP). Next press the RECORD and PLAY buttons down together. If you are not using leaderless tape, pull out the MIC plug for about 5 seconds. (This will make sure that you begin recording on blank tape.) Now you are ready to record the procedures. Simply respond **T** **ENTER** to the **SAVE:** prompt:

```
LOGO: SAVE: T
```

When the recording is done, the BREAK mode prompt will be displayed again. If a number and a question mark appear after the **T**, then the procedures were not saved properly, so try again.

Disks must be formatted before use with Color LOGO. Disks can be formatted using the DSKINI0 command when the computer is running BASIC. However, Color LOGO does not use the BASIC directory, so once a disk is used for storing Color LOGO programs, it must not be used with BASIC programs.

If you do not have a formatted disk, save your procedures on tape (as described above) and turn off the Color Computer. Then turn the computer back on. The computer will be running BASIC now. Insert a blank disk (remember, label towards the latch). Then type **DSKINI0**, and press **ENTER** . If you get an error message, remove the disk, center it in its paper wrapper, and try again. Because it is always a good idea to keep duplicates of programs on disk, format a second disk while you're at it. Once you get the **OK** response, you can reload Color LOGO as covered in Chapter 2 and reload your procedures from tape as described below.

A disk for Color LOGO programs is divided into 16 modules. Each module occupies 2 tracks or 36 sectors, so the capacity of a module is 9K. It is doubtful whether you will ever come very close to filling a module with a set of procedures because LOGO procedures tend to be relatively short. (If you do fill the 9K, the editor will refuse to accept additional text.) The modules are indexed by the letters A through P. To save all the procedures in memory on the disk type one of the letters A through P in response to the BREAK mode prompt

LOGO: SAVE:__

You must press ENTER to start the save. While you're at it, remove this disk, insert the second formatted disk, and save the procedures on the same module (again press S, the module letter, and ENTER.

We have on rare occasions had trouble storing programs on the disk. The trouble seems to be related to the relative positions of the disk or the ribbon cable and the TV set. Whatever, after saving on disk we try reloading the module immediately. If the load is not successful, we get an error message (a 6 followed by a question mark), but the procedures are still in memory, and so we can try another save without retyping the procedures.

We suggest that you immediately start an index card for any new disk of Color LOGO programs. No diskette labels are likely to be large enough for you to indicate all the procedures on a disk, and it is a poor idea to write on labels once they are on the disk. Make sure that you write the disk name on both the card and the disk label before they have a chance to get separated. One reasonable format is:

Disk Name ________________

A	__________	I	__________
B	__________	J	__________
C	__________	K	__________
D	__________	L	__________
E	__________	M	__________
F	__________	N	__________
G	__________	O	__________
H	__________	P	__________

Loading programs from cassette and diskette is also simple. Again it is necessary to be in BREAK mode. In response to the BREAK mode prompt, press L. The prompt then will read

LOGO: LOAD:__

The responses here are exactly the same as for **SAVE**: use letters A through P to load a disk module, and T to load from tape. In both cases pressing ENTER will start the process. Of course with the cassette tape you will have to have the volume set to about 5, have the tape rewound, and have the PLAY button depressed before pressing ENTER.

If you have a printer for your Color Computer, you can print all the procedures in memory. Again it is all or nothing, except that you can interrupt the printing by pressing BREAK without damaging or losing the programs in memory. To print, connect the printer as described in the Owner's Manual; load the paper and turn on the printer. From BREAK mode, enter P for single space, or Q for double space, and the procedures in memory will be printed. If for some reason you want to eliminate the line feed at the end of any line (thus using a larger portion of the paper width), get into EDIT mode and insert an **@** character at the end of every line for which you want to eliminate the carriage return and line feed. (To place an **@** character in a line in EDIT mode, you'll need to press the @ key twice.)

There remains the question of saving results, the pictures on the screen. We find that the best way is to take pictures (this is the only way to get color at any reasonable price). To avoid false patterns due to interactions of the camera shutter with the video display, we recommend a shutter speed of 1/2 second. Use a tripod and a cable release for the camera. The lens setting is somewhat dependent on the brightness setting of the TV, and of course on the film speed. A good starting point is to set medium brightness on the TV and use a lens opening of about f8 with film speed of 100 ASA. The reds are likely to come out rather brownish, and commercial developers are likely to overexpose prints with large dark backgrounds. However, the illustrations in this manual are typical of what can be done without much trouble. You will minimize distortion if you use a telephoto lens.

This chapter is short because there is very little complication in saving, loading, and printing programs. Remember that you have a very large procedure space available, so if you are starting a new project and think you may want to use some pieces of a previous project, load the previous project before you start. You can always delete what you don't need from memory, and you have plenty of space.

10. RECURSION

In the Color LOGO language, any procedure can call any procedure. When the procedure calls itself, we have a very powerful logical structure called recursion. One clever example of recursion was given by Hofstader in his book *Godel, Escher, Bach.*

Hofstader's Law: It always takes longer than you expect,
even when you take into account Hofstader's Law.

There are actually two types of recursion. We'll start with the easier one: recursion where the call is the last statement of the procedure. As usual, it is easiest to look at examples. Recursion can be used in place of the **REPEAT** statement.

```
TO CIRCLE
   FD 3   LT 10
   CIRCLE
END
```

When we run **CIRCLE** the turtle moves forward three steps and turns. Then **CIRCLE** is called, which causes the turtle to move forward three steps and turn, etc. In principle this process could continue forever. However, every time a procedure is called some memory is used up. Eventually the memory is all used up and we get the message,

```
"MY MEMORY IS TOO FULL."
```

Try it.

So although recursion can be used exactly like **REPEAT**, there are some disadvantages to doing this. We have to find some way of stopping the computer, or it will run out of memory. There are also some great advantages to using this type of recursion. The following program appears in all LOGO books and manuals.

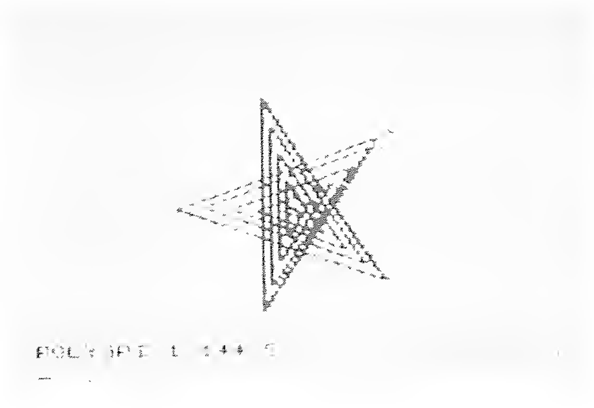
```
TO POLYSPI  :SIZE  :ANGLE  :STEP
   FD  :SIZE
   RT  :ANGLE
   POLYSPI  (:SIZE + :STEP)  :ANGLE
                :STEP
END
```

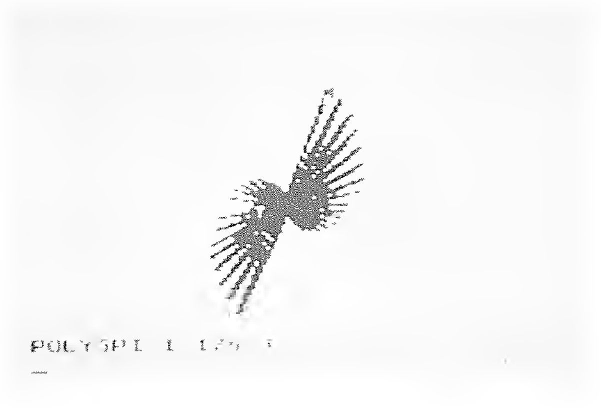
This procedure is so much fun to play with that I think you should do so before we get involved in any explanations. One suggestion before you start: the figures created are likely to outgrow the screen long before the memory runs out. The wrap-around feature of the screen will then lead to some striking but puzzling effects. To start with, let's prevent wrap-around. Enter RUN mode and type

```
NOWRAP
```

Then try a variety of runs, for example

```
POLYSPI 1 90 1
POLYSPI 1 90 5
POLYSPI 1 120 3
POLYSPI 1 122 3
POLYSPI 1 144 5
POLYSPI 1 145 1
POLYSPI 1 176 3
```



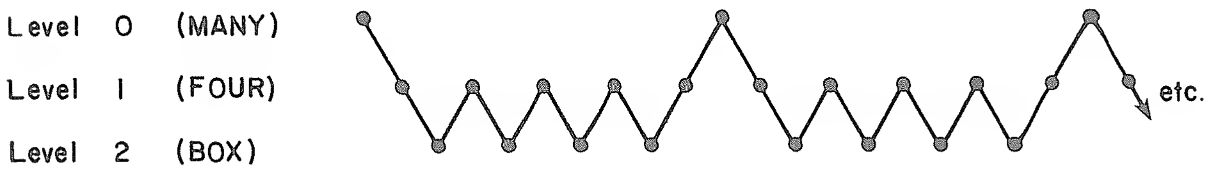


If you then want to see what happens when the computer allows wrap-around, type

```
WRAP
```

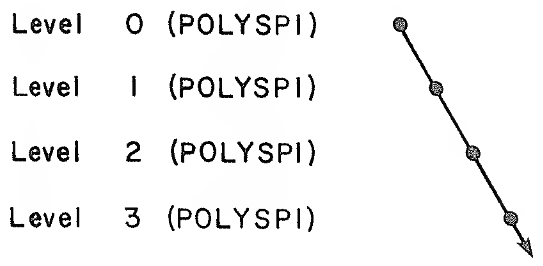
and try some more runs.

Now let's try to figure out what is going on with this **POLYSPI**. It is useful to think of a Color LOGO program in terms of levels: the main program is a procedure at level 0, a subprocedure called from level 0 is at level 1, a subprocedure called from level 1 is at level 2, etc. The operation of a program like **MANY** can be diagrammed as



The transitions down and up between levels 0 and 1 are controlled by the **REPEAT** statement in **MANY** (down to level 1) and the **END** statement in **FOUR** (up to level 0). The transitions down and up between levels 1 and 2 are controlled by the **REPEAT 4** statement in **FOUR** (down to level 2 four times) and the **END** statement in **BOX** (up to level 1).

In a program like **POLYSPI** the path is actually less complex.



The transitions down are controlled by the statement

```
POLYSPI (:SIZE + :STEP) :ANGLE :STEP
```

There are no transitions back up because the procedure never reaches the **END** statement. Thus the computer sinks down level after level until it finally runs out of memory.

POLYSPI gives interesting figures because of the difference in the values of the variables at the different levels. We begin the program with the command

```
POLYSPI 1 9Ø 5                    (Level Ø)
```

The recursive call of the procedure is (when we substitute the current values of the variables)

```
POLYSPI 6 9Ø 5                    (Level 1)
```

Therefore at level Ø the **FD** command is for length 1 and at level 1 the **FD** command is for length 6. The pattern continues

```
POLYSPI 11 9Ø 5                   (Level 2)
POLYSPI 16 9Ø 5                   (Level 3)
```

until the turtle runs off the screen (**NOWRAP**) or the computer runs out of memory (**WRAP**).

Another procedure of this type is called **INSPI**. In **INSPI**, the angle is incremented recursively.

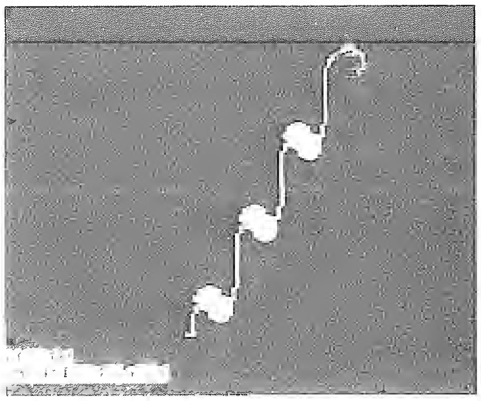
```
TO INSPI :SIZE :ANGLE :STEP
  FD :SIZE
  RT :ANGLE
  INSPI :SIZE (:ANGLE + :STEP)
  :STEP
END
```

Again this is one to play with. Try

```
INSPI 1Ø 9Ø 5
INSPI 1Ø 9Ø 3
INSPI 7 5 3
```

(do **SX 8Ø SY 14Ø** first to get it all on the screen)

```
INSPI 3 5 2
```



(16K users won't see all of this picture.)

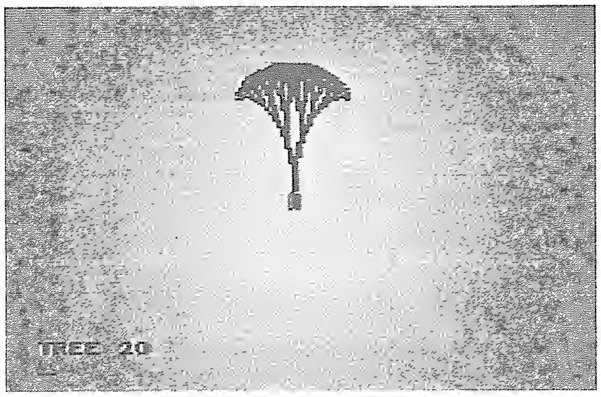
Incidentally, if you want to get rid of the type at the bottom of the screen, you can use a procedure like the following:

```
TO TEST :SIZE :ANGLE :STEP
   CLEAR
   INSPI :SIZE :ANGLE :STEP
END
```

The analysis of **INSPI** is so similar to that of **POLYSPI** that we won't bother to give it.

We now turn to the more complex type of recursion, recursion where returns to the higher levels are actually made. Another popular program in turtle geometry is called **TREE**.

```
  TO TREE  :N
     IF :N<2 (STOP)
     FD :N
     RT 15
     TREE (3*:N/4)
     LT 30
     TREE (3*:N/4)
     RT 15
     BK :N
  END
```



TREE 20

The line numbers are not part of the procedure; they are there for reference in our discussion of the procedure. Enter this procedure (without the line numbers, of course) and try running it with a value of 2Ø to 3Ø for :**N**. You might try making the numerical factors in the two calls of **TREE** (lines 4 and 6 of the procedure) somewhat different, thus producing an asymmetrical tree. You also could try changing the angles, but notice that the sum of the two right turns is equal to the left turn.

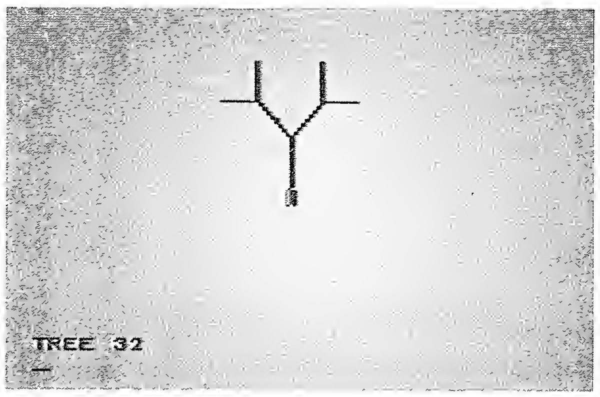
Now let's try to understand the program. We have introduced two new ideas in line 1. The first is the conditional **IF**. The **IF** must be followed by an expression which has a truth value. In **TREE** the expression is

```
:N < 2
```

If the current value of :**N** is less than 2, then this expression is true and the rest of the statement will be executed. If the current value of :**N** is 2 or greater, then this expression is false and the rest of the statement will be skipped. The rest of the statement is placed in parentheses; it may consist of many commands, and it may extend over many lines.

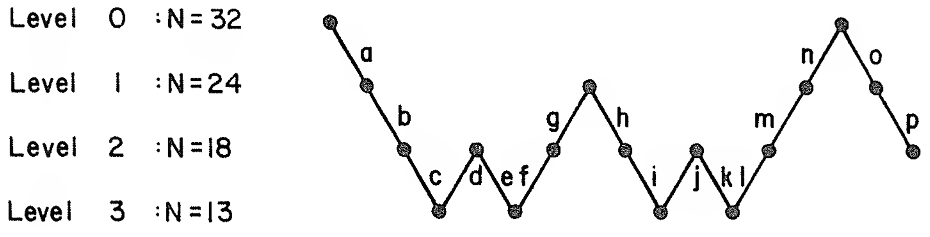
The second new item in line 1 is the control statement **STOP**. The **STOP** statement ends a procedure. **STOP** has the same effect as an **END** statement, but an **END** statement can appear only at the end of a procedure.

It will be easier to understand **TREE** if we make some changes to simplify it. Change the condition in line 1 to :**N <18**, and change the right turns (lines 3 and 7) to **45** and the left turn (line 5) to **9Ø**. Then run **TREE 32** to get a simpler tree.



```
TREE 32
_
```

The following diagram and table outline the operation of **TREE**.



STEP	:N	START	END	LINES
a	32	0	R45	1-4
b	24	R45	R90	1-4
c	18	R90	R135	1-4
d	13	R135	R135	1
e	18	R135	R45	5,6
f	13	R45	R45	1
g	18	R45	R90	7-9
h	24	R90	0	5,6
i	18	0	R45	1,4
j	13	R45	R45	1
k	18	R45	L45	5,6
l	13	L45	L45	1
m	18	L45	0	7-9
n	24	0	R45	7-9
o	32	R45	L45	5,6
p	24	L45	0	1-4

etc.

First look at the diagram. The program tries to reach the lowest level of the procedure until forced to rise in level by the **STOP** or the **END** statement. By a method which we will explain in Chapter 14 we can single-step the program by running it as follows. Type

```
HATCH 1  TREE 32
```

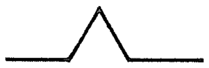
Now every time you press ENTER, one command in the procedure will be executed. For example, when you start (step a), the first ENTER executes **IF :N<18**, the second ENTER executes **FD :N** (which you can see), the third ENTER executes **RT 45** (again visible), and the fourth ENTER executes the recursive call. You can follow the table exactly, as long as you realize that the statement

```
IF :N<18 (STOP)
```

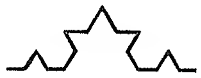
is one statement if the condition is false, but two statements if the condition is true (execute the **STOP**). A step in the table is counted as all the lines executed from when a level is entered until the level is exited. A level can be exited in three ways: by a **STOP** or an **END**, which completes the procedure at that level and goes up; or by a call of another procedure, which leaves the current procedure incomplete and goes down.

Instead of giving an elaborate account in words of what is happening, we recommend that you run the program in single-step mode and follow the table and the program in parallel. If you get confused, start over and try again. Recursion is a little complex, but it is so powerful that it is worth the effort to understand.

The next program draws a figure which is called a fractal. A fractal is a figure which looks the same no matter what magnification is used to view it (of course we are limited by the screen resolution here). In this example we'll start with the basic shape

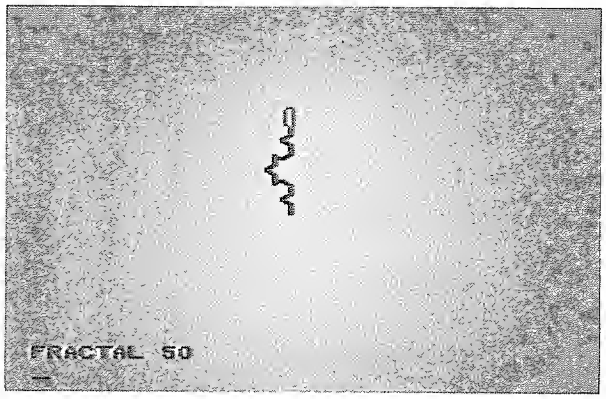


The idea is that each of these four lines should be made up of that same shape. At one additional level of detail that gives



Each of the lines in this drawing is in turn made up of four lines with the basic shape, etc. This is a place to use recursion.

```
TO FRACTAL  :N
   IF :N <15 (FD :N STOP)
   FRACTAL (:N/3)
   LT 60
   FRACTAL (:N/3)
   RT 120
   FRACTAL (:N/3)
   LT 60
   FRACTAL (:N/3)
END
```

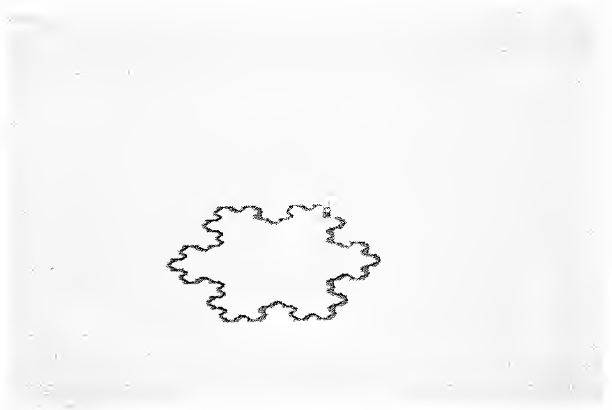


```
FRACTAL 50
_
```

Running **FRACTAL 50** will show the pattern. (You may want to enter **RT 90** and **SX 0** before running **FRACTAL 50** to turn it the way we've drawn it in the manual.) Notice that the pattern divides the whole length into thirds; that is why we divide by 3 on the recursive calls. However, the resulting length will change somewhat depending on the conditional IF statement because of the roundoff loss in integer arithmetic. You'll see that if you change the conditional to something finer—say, :**N** < **4**. That pattern looks like the edge of a snowflake. Why not make it into something six-sided?

```
TO FLAKE :N
  CLEAR
  REPEAT 6 (FRACTAL :N  RT 60)
END
```

You may have to play with the starting position (**SX** and **SY**) and the size to get a nice figure without wrap-around. You may also prefer the figure you get when **FLAKE** is made to draw three sides at 120 degrees.



Other variations are possible. We can replace **FD :N** in the **IF** statement with a more elaborate series of commands. A few examples follow.

```
TO FLAKE :N
  CLEAR
  SX 50
  SY 50
  REPEAT 3 (FRACTAL  :N  RT 120)
END
```

Replace the conditional statement in **FRACTAL** with

```
IF :N<9 (FD :N/4  RT 80
         FD :N  LT 160  FD :N
         RT 80  FD :N/4  STOP)
```

and try

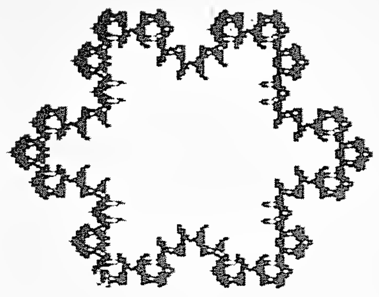
```
FLAKE 150
```

Another variation is

```
IF :N<9 (FD :N/4  RT 80
         FD 2*:N  LT 160  FD 2*:N
         RT 80  FD :N/4  STOP)
```

and try

```
FLAKE 150
FLAKE 70
```

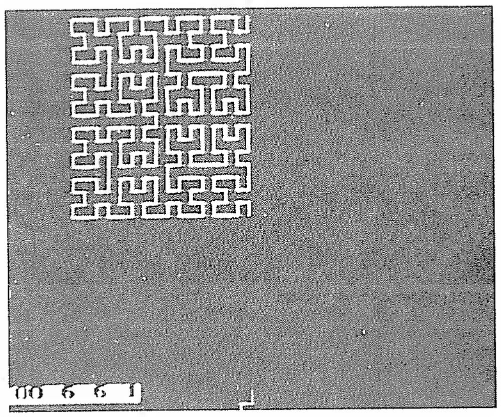


Recursion can be used to draw endless space-filling patterns. The following example is typical.

```
TO FOO :SIZE :LEVEL :PARITY
  HT
  IF :LEVEL = 0 (STOP)
  LT :PARITY*90
  FOO :SIZE  (:LEVEL-1)
        (:PARITY*-1)
  FD :SIZE
  RT :PARITY*90
  FOO :SIZE  (:LEVEL-1)  :PARITY
  FD :SIZE
  FOO :SIZE  (:LEVEL-1)  :PARITY
  RT :PARITY*90
  FD :SIZE
  FOO :SIZE  (:LEVEL-1)
        (:PARITY*-1)
  LT :PARITY*90
END
```

An appropriate set of numbers is

```
FOO 6 6 1
```



In this chapter we have tried to give you some ideas for recursive programs. However, we have just scratched the surface of the designs that are possible. Go ahead and experiment, and let others know if you discover any new beautiful designs.

11. DOODLE MODE — PROCEDURES WITHOUT TYPING

Color LOGO provides a way to enter graphic procedures into the computer without typing the turtle commands like **FORWARD** and **RIGHT**. The reason for including this feature is to provide a way for children who are not yet able to read or not yet able to type reliably to use the language and to benefit from the practice in structured thinking that the language offers. The features of DOODLE mode are arranged with that audience in mind. Obviously the children are not going to be able to read the manual, so a parent or teacher will have to assist them in learning. In this and the next chapter we will teach the assistant the mechanics of two ways to use Color LOGO with children; actual suggestions for activities with the children are gathered together in Chapter 13. In this chapter we will cover the mechanics of DOODLE mode.

The idea of DOODLE mode is that a minimum set of turtle commands can be entered by single keystrokes from the keyboard. Before proceeding, you should get the plastic overlay for the keyboard which was supplied with the Color LOGO package. This overlay fits over the top row of the keyboard. The symbols on the overlay show the meanings of the numeric keys in DOODLE mode (see Appendix 1, page 107). These symbols will appear at the bottom of the screen in DOODLE mode when the keys are pressed, and they will appear in the procedures that you create in DOODLE mode.

DOODLE mode is entered from RUN mode by pressing the @ key. When the @ key is pressed, an = appears at the bottom of the screen. This is the indication that it is time to name the procedure you are creating. Simply type the name you want to use; the name can be as simple as a single letter or number. After typing the name, press ENTER. The computer is now in DOODLE mode and the top row of keys has its new meaning.

The meanings of the keys are

(1) **CLEAR**	(2) **HOME**	(3) **PENUP**	(4) **PENDOWN**
(5) **RT 45**	(6) **LT 45**	(7) **FD 1**	(8) **FD 1Ø**
(9) **RT 15**	(Ø) **LT 15**		

Try each of the keys in turn. Of course **1** (**CLEAR**) will clear the screen, so you won't have much time to see that one. Note the correspondence between the symbols on the keys, the symbols on the bottom of the screen, and the action of the turtle. Remember that a turn of 15 degrees is not enough to turn the turtle shape, but three 15 degree turns in a row is enough to produce a visible turn.

OK, now that you have a bit of the idea of DOODLE mode, let's try to make something useful. To get a fresh start, exit DOODLE mode by pressing BREAK. This puts you in BREAK mode. Clear the procedure space by pressing SHIFT CLEAR. Get into the RUN mode (press R) and then get into the DOODLE mode (press @). The reason that we always get into the DOODLE mode from RUN mode is that we may want to draw patterns on the screen in RUN mode for the child to interact with or copy in DOODLE mode. Notice that the screen is not cleared when we enter DOODLE mode. Now let's name the procedure we are creating "**S**" by typing an S and pressing ENTER.

Begin by drawing a box using the top row of keys. When the box is completed, exit DOODLE mode by pressing the BREAK key. Then get into the RUN mode and run the procedure **S**. The only difference between this and the procedure we earlier called **BOX** is that **S** is a little slower. To see that **S** actually exists as a procedure, get into EDIT mode and look at procedure **S**. Notice that it is there with exactly the symbols used in its definition.

We can actually edit **S** in EDIT mode just as we edit any other procedure. For example, we can add a diagonal to the box in several ways. We can add the type of turtle commands we are already familiar with, like

```
RT 45
FD 6Ø
```

This shows that DOODLE mode turtle commands can be mixed with regular turtle commands. However, in some sense this way of editing defeats the purpose of DOODLE mode, as the child is not likely to be able to understand the change. To keep it understandable for the child, we edit using the DOODLE mode symbols in EDIT mode. Each DOODLE mode symbol can be made by pressing @ followed by the appropriate key. Thus we could insert the above instructions for a diagonal by the series of keystrokes

```
@5@8@8@8@8@8@8
```

That is, an **RT 45** (**@5**) followed by six **FD 1Ø**'s (**@8**). Try it.

Obviously this kind of editing would be most useful in a cooperative project where the child was using DOODLE mode and the helper was using EDIT mode. A more likely type of error correction or editing is to make changes during the doodling process. For example, "I should not have taken that last step forward," or "I should not have turned that far." To see how to handle this, get into DOODLE mode (BREAK, R, @) and enter a new name, say B. Now doodle out a box, but go one step too far on the last side. Left-arrow (backspace) will now erase the last step in the procedure. Unfortunately it also erases the listing of the procedure from the screen and clears the screen, but it does redraw the shape with the last step now eliminated. Try eliminating step after step by repeated use of the left-arrow key. The edited version of the procedure is stored in the memory in correct form and can be seen in EDIT mode.

Another type of editing the child may wish to do is to add on to the end of a previous procedure. There is no simple way to do exactly that, but it is easy to produce the same effect. Get into RUN mode and run the current version of the procedure. That will draw the shape on the screen. Get into DOODLE mode and give a second name. Notice that the turtle is at the home position instead of at the end of the shape. Start the new procedure with **HOME** (key 2), raise the pen (key 3), move to the end of the shape, and lower the pen (key 4). Now you are ready to proceed with completing the shape. To get the whole shape while running, either run the two procedures in sequence or, in EDIT mode, remove the **END** statement from the first procedures and the **TO** name statement from the second procedure. If you do the latter, you can remove all the turtle commands from **HOME** to **PENDOWN** at the start of the second procedure, as well.

Thus far we have limited ourselves to horizontal, vertical, and 45-degree lines. What about other angles? It might be useful for you to reread the discussion about circles in Chapter 5 at this point. There we saw that with a step forward of one unit the only angles which were possible were 45 degrees apart (Ø, 45, 9Ø, ...). In DOODLE mode the only steps forward possible are one unit and ten units. We've already said that the only turns that make any difference with one unit forward are turns which are multiples of 45 degrees. The only turns which make sense with ten units forward are multiples of 15 degrees. (This is not as obvious or as absolute as the one unit forward case, so if you don't see the problem, try the following procedure.

```
TO TEST  :ANGLE
   RT 9Ø  +  :ANGLE
   REPEAT 2  (FD 1Ø)
END
```

The design decision was that 15 degrees was the smallest divisor of 45 degrees that gave a smooth line at an angle.)

If you want to make a line at some other angle in DOODLE mode, you can. It just requires more keystrokes. The combination of **FD 1** operations and either no turn or a turn of 45 degrees will allow the drawing of a line at any angle. For example, a very small angle could be drawn by repeating the series--**FD1Ø** (key 8) followed by nine **FD 1**'s (key 7), an **RT 45** (key 5), an **FD 1**, and an **LT 45** (key 6)—the desired number of times. This sounds awkward, but remember that it will be necessary very seldom. Most young children will find sufficient accuracy with turns of 15 and 45 degrees.

DOODLE Mode characters are not available on printers. If you try to print a procedure created in DOODLE mode, the special characters will appear as lower case letters. Thus it is impractical to print DOODLE mode procedures. Also notice that the **@** character has a special meaning in EDIT mode. If you want to include an **@** character in a line of a procedure to prevent a line-feed when printing, you must press $\boxed{@}$ twice in a row.

12. ONE KEY DOODLING

The idea of DOODLE mode can be extended to an open-ended set of single keystroke operations if we give up the ability to store and edit the child's input as a procedure. This requires a set of procedures which we hereafter refer to as the OK Set (One Key Set). To start we define a set of procedures with single-character names. Then pressing the single key and ENTER will call forth the desired action. This is easier shown than described.

The first step is to define a set of procedures which match the individual keys in DOODLE mode. Because we are not going to save or edit the procedures, we do not bother to draw the special symbols at the bottom of the screen (although if there was some reason to have them we could draw them with turtle commands). Clear the memory and enter the following procedures.

```
TO 1
  CLEAR
END

TO 2
  HOME
END

TO 3
  PU
END

TO 4
  PD
END

TO 5
  RT 45
END

TO 6
  LT 45
END

TO 7
  FD 1
END

TO 8
  FD 1Ø
END

TO 9
  RT 15
END

TO Ø
  LT 15
END
```

This set of procedures will allow the child to move the turtle freely around the screen in RUN mode using the keys he or she already knows from DOODLE mode.

The advantages of this approach become evident when we add to the list of procedures. The following procedures are typical.

```
TO T
  SH Ø  HT  PD  FD 8
  RT 15Ø  FD 15
  TRI 15
  SH Ø  FD 5  PU  ST
END
```

```
TO TRI :SIDE
  IF :SIDE<2 (STOP)
  REPEAT 3 (RT 120 FD :SIDE)
  TRI (:SIDE-2)
END
```

T will draw a triangle. You might wonder why **T** is so elaborate; after all we could use the following to draw a triangle.

```
TO QUICKT
  REPEAT 3 (FD 15 RT 120)
END
```

The problem here is that the orientation of a triangle drawn by **QUICKT** will depend on the prior heading of the turtle. For the applications we have in mind we want all the triangles to have one vertex pointing up (**SH 0**). We also want to color-in the triangle, and we want to draw the triangle around the turtle's starting position. Therefore, we use the procedure **TRI** and we move forward eight units before starting the triangle; hiding the turtle gains speed. **TRI** uses recursion to make a filled-in triangle. To get complete filling-in we must start at the correct vertex of the triangle (this is not obvious, but is a consequence of the way in which the Color Computer produces color in high resolution). Thus the line **RT 150 FD 15** in **T** moves us to a different vertex. You might try replacing this line with **RT 30** to get a striped triangle, and then make **TRI** read **REPEAT 4 (RT 129 FD :SIDE)** to get an even more interesting pattern. The commands in **T** after **TRI 15** return the turtle to the starting position with a heading of **0** degrees.

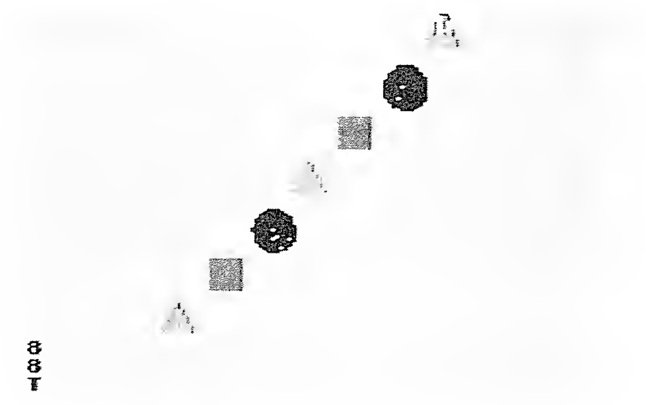
A similar set of procedures can be used to define a square and a circle.

```
TO S
  SH 45 HT PD FD 10 RT 45
  SQU 14
  RT 135 FD 10 SH 0 PU ST
END

TO SQU :SIDE
  IF :SIDE<2 (STOP)
  REPEAT 4 (RT 90 FD :SIDE)
  SQU (:SIDE-1)
END

TO C
  SH 0 HT PD FD 8 RT 90
  REPEAT 15 (FD 4 RT 24)
  RT 22 MAKE :X 7
  REPEAT 7 (CIR :X RT 90 FD 1
            LT 90 MAKE :X :X-1)
  RT 135 FD 2 SH 0 PU ST
END
```

```
TO CIR  :STEP
   REPEAT 8  (FD  :STEP  RT 45)
END
```

The last line before the **END** in **C** restores the turtle to the original position. It is not easy to compute what moves are necessary to reach the original position, so we do it by experiment. Run **C** immediately upon entering **RUN** mode, so that the starting position for the turtle is at the home (128,96) position. When the procedure has finished, do an **SX 128** and an **SY 96**. Watch which way the turtle moves, if at all. From those moves we can tell what changes must be made in the last line of **C** to restore the turtle to the original (home) position.

With these procedures the child can move the turtle around the screen using the number keys, and the child can produce triangles by pressing **T**, squares by pressing **S**, and circles by pressing **C**. But, I hear you say, this is for children who don't know the letters. We suggest that you cover the selected keys with small adhesive labels on which the symbols have been drawn. In this example this would mean putting a label with a triangle on the T key, a label with a square on the S key, and a label with a circle on the C key. Of course you could use any other keys instead by renaming the procedures.



As in DOODLE mode we want to have some way to erase mistakes. The way to do this is to redraw the shape with the pencolor set to the background color. We also have to pick a way for the child to control the erase. One possibility is to use double presses of the same key to specify erase. With a minor name change, then, we have the procedures

```
TO T                              TO TT
   PC 1                              PC 3
   T1                                T1
END                               END

TO T1
   SH Ø  HT  PD  FD 8
   RT 15Ø  FD 15
   TRI 15
   SH Ø  FD 5  PU  ST
END
```

The fact that we return the turtle to its original position makes this erase possible.

Similar changes in **S** and **C** give

```
TO S                              TO SS
  PC 2                              PC 3
  S1                                S1
END                               END

TO C                              TO CC
  PC Ø                              PC 3
  C1                                C1
END                               END
```

We have not bothered to reprint the original versions of **C** and **S** which must be renamed **C1** and **S1**.

While we are at it we should allow for double keystrokes of the DOODLE mode commands. One example should be sufficient.

```
TO 77
  PC 3  BK 1
END
```

We are in effect building a special language consisting of one-keystroke commands. Because of the low frustration tolerance of the audience that we are building the special language for, it is especially important to make the language "user-proof." To do that, we should define a procedure for every other key on the keyboard. The procedures are

```
TO A
TO B
TO D
...
END
```

Note that we skipped **C** because it is actually used. Note also that it is not necessary to have individual **END** statements for each procedure because the following **TO** statement automatically ends each procedure. These procedures actually prevent the message

```
I DON'T KNOW HOW TO ...
```

if the child accidentally presses an unlabeled key.

In this chapter we have introduced the idea of building shapes or complex picture elements which the young child can call forth with single keystrokes. The examples we have given are simple, but the only limit to what is possible is your time and imagination. Let's now start thinking about ways to use these tools with very young children.

13. USE OF DOODLE MODE AND OK SET

In the previous chapters we covered the operations of the DOODLE mode and the OK Set. What is possible and what is worthwhile are two separate questions. In this chapter we will pass along some suggestions. Our suggestions are aimed at the adult who is working with small children. We have collected ideas from a number of sources. However, we should make it clear that, because Color LOGO offers possibilities for working with much younger children than could be reached previously, no one at this time really knows what is possible or what is most beneficial. Also remember that this is a user's manual for a computer language, not a textbook on early childhood education. Don't be hesitant to question our suggestions, and don't hesitate to try out new ideas.

Perhaps the best way to start with very young children is to let them play. By play we mean to allow them to explore the effects of the various keys. If the children are very young, this will take quite a bit of time. If you've changed the shapes available in the OK Set since the last session, then you should give the child another chance to explore the new set of keys. Keep in mind that a child's attention span is not as long as yours, so don't try to prolong the sessions. Our own first ideas for Color LOGO grew from an effort to create something for a four-year-old to do because he wanted to be like Dad and "work on the computer." This suggests that another way to start is to master DOODLE mode yourself and to prepare a set of procedures for the OK Set. Then you'll be ready when an interested face appears at your shoulder some evening.

The users of LOGO have had consistent success with one technique for getting children started. They repeatedly relate the turtle commands with body movement. That is, ask the child to play turtle and ask them to keep track of the turtle movements they make. Thus, if the task is to draw a box, the child is asked to walk in a box-shaped path and then to tell the turtle what he or she did. The success on which this recommendation is based comes from work with somewhat older children, so it may not be quite as effective with the pre-reading group. It might help to give them objects to actually walk around to make the shape less abstract.

One heavily used technique in early childhood teaching is to ask the student to copy something. A book titled *Mathematics Their Way* by Mary Baratta-Lorton (Addison-Wesley Publishing Company, 1976) makes use of this technique for beginning mathematics and is a rich source of ideas for DOODLE mode projects. Basically, the approach is to write a procedure in the conventional way which will draw a figure or shape. This can be placed on the screen in RUN mode. The child is then asked to copy, complete, fill in, invert, rotate, or in some way proceed with reference to the figure on the screen. If the procedure the child develops while doing this is likely to be worth keeping for future use, then the child should be working in DOODLE mode. If it is not worth keeping, or if it requires the more complex shapes, then the child should be working with the OK Set of procedures.

Let's turn now to some specific activities. A large number of exercises could be built around the idea of pattern continuation and generalization. These activities are best suited to the OK Set, so the following procedures should be added to that set. One of the simplest types is a pattern which can be imposed on a line of dots. First we draw two parallel lines of dots.

```
TO DOTS
   CLEAR
   HT   RT 90
   SX 5  SY 150
   LINE-OF-DOTS
   SX 5  SY 50
   LINE-OF-DOTS
END

TO LINE-OF-DOTS
   REPEAT 20  (DOT  FD 12)
END

TO DOT
   FD 1  PD  RT 90  FD 1
   REPEAT 4  (RT 90  FD 2)
   PU
   BK 1  LT 90  BK 1  PU
END
```

DOT is a bit more elaborate than we need right now, but later on we'll want to be sure that the dot is centered on the starting point. We'll use **DOT** in later examples. Next we draw some very simple repeating pattern on the upper row of dots. We'll use **DOTS** in the pattern drawing procedure. Several examples follow.

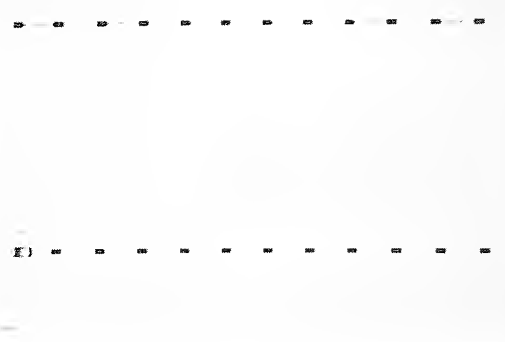
```
TO PATTERN1
   DOTS
   SX 5  SY 150
   REPEAT 10  (PD  FD 12  PU
                  FD 12)
   SX 5  SY 50  ST  PD
END
```

Try running **PATTERN1**. The idea is that the child is to reproduce the pattern on the upper set of dots on the lower set, and then, after that is mastered, the child is to give an equivalent pattern with some other shapes. Before you try this with a child, try it yourself! Try to copy the pattern on the lower row of dots. You'll find that it is more difficult than necessary. It is needlessly difficult to move the turtle the correct number of units (12 as the procedure is written). We can make the exercise much less bothersome by some minor adjustments. Notice that these adjustments in no way detract from the point of the exercise, which is to recognize and continue the pattern.

```
TO LINE-OF-DOTS
   REPEAT 12  (DOT  FD 20)
END
```

The change to **FD 20** means that the child can connect dots with two keystrokes (key 8 producing **FD 10** on each stroke). We have to adjust **PATTERN1** as well.

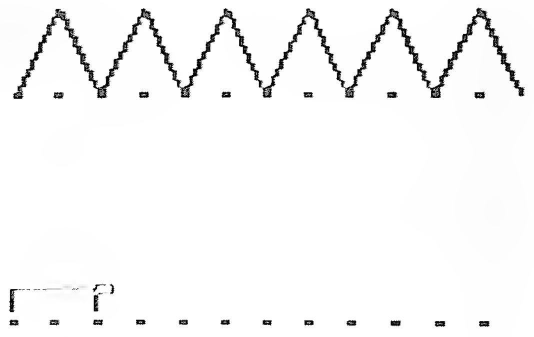
```
TO PATTERN1
   DOTS
   SX 5  SY 150
   REPEAT 6  (PD  FD 20  PU
                 FD 20)
   SX 5  SY 50  ST  PD
END
```



You may be wondering we didn't just give you the final versions immediately. The point is that we hope you will try creating your own exercises, and we want you to see that a little attention to detail can make the exercises much more effective. Especially with very young children, be sure to try out the task to check the difficulty level before they are around. This is supposed to be fun as well as instructive, and not a new source of frustration.

The same pieces can be used for a slightly more difficult exercise.

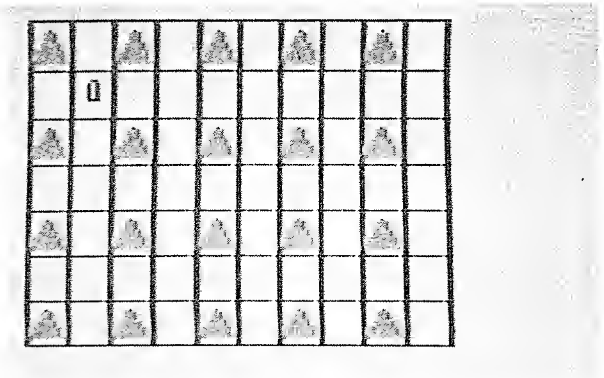
```
TO PATTERN2
   DOTS
   SX 5  SY 150
   REPEAT 6  (PD  LT 60  FD 40
                 RT 120  FD 40
                 LT 60)
   SX 5  SY 55
   LT 90  FD 10  RT 90  FD 40
   RT 90  FD 10  BK 10  LT 90
   ST
END
```



Here the task is to reproduce the indicated pattern by continuing the shapes started on the lower line. Because the two shapes forming the pattern are different, the focus is on the shape rather than straight copying.

We may as well make use of some of the fancier shapes that we have defined in the OK Set. The following is another example of complete-the-pattern, but one which is visually more interesting.

```
TO PATTERN3
   MAKE :X 0   MAKE :Y 50
   CLEAR  HT
   REPEAT 10
      (REPEAT 7  (SX  :X  SY  :Y
           SQUARE   MAKE :Y  :Y+20)
       MAKE :X  :X+20   MAKE :Y 50)
   MAKE :X 11   MAKE :Y 58
   REPEAT 4
      (REPEAT 5  (SX  :X  SY  :Y
           T   MAKE :X  :X+40)
       MAKE :X 11   MAKE :Y  :Y+40)
   SX 31  SY 158  ST
END

TO SQUARE
   REPEAT 4  (FD 20  RT 90)
END
```



The child's task is to complete the pattern by moving the turtle and by pressing the key with the triangle. Many other variations on this theme are possible.

The **PATTERN3** procedure makes heavy use of the **MAKE** statement, and we have not discussed that before. The **MAKE** statement changes the variable following the **MAKE** to the value given by the next expression or number. For example

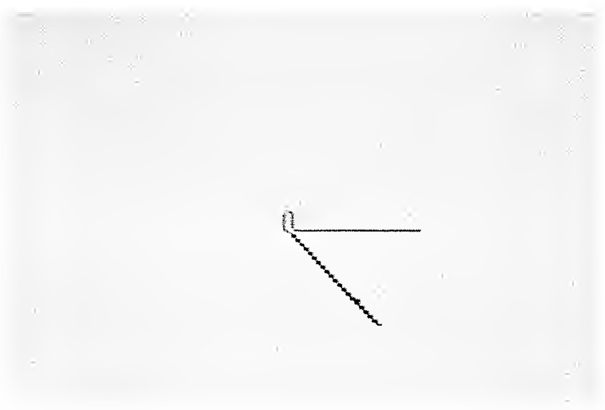
```
MAKE  :X  :X + 40
```

replaces the starting value of **:X** with a value which is 40 greater.

Of course not all the tasks using the triangles, squares, and circles need to be directed towards specific goals. Ask the child to create a design, or to create a border to the screen.

Another group of projects can be based on completion of design. The screen can be thought of as consisting of four quadrants divided at the home position. The idea is to have the turtle draw a pattern in one quadrant and to have the child complete the pattern in the other three quadrants. Either DOODLE mode or the OK Set can be used here. If you've included the erase procedures for the DOODLE turtle commands in the OK Set, then that set is preferable. A simple pattern is

```
TO PATTERN4
   CLEAR
   RT 9Ø
   REPEAT 2  (FD 6Ø  SX 128  SY 96
                  RT 45)
   HOME
END
```

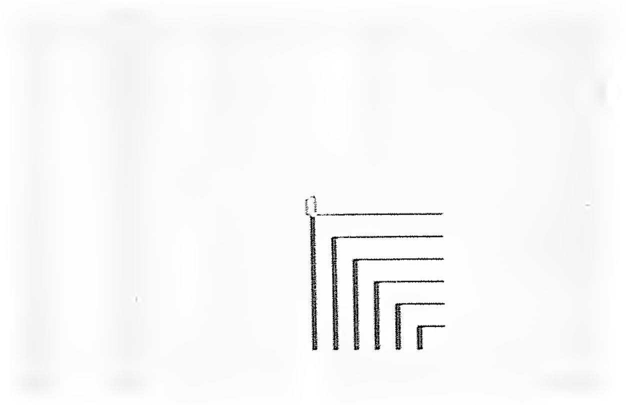


We have written the procedure so that it is easy to add lines to make a more elaborate pattern. However, we recommend that you restrict the patterns to those using angles which are easy to produce in DOODLE mode (that is, multiples of 15 or 45 degrees). We reset the turtle to the home position with the **SX** and **SY** instead of with **HOME** so that the turtle heading is preserved. Again the **FD** should be some multiple of 1Ø to minimize the number of keystrokes needed.

This gives a more complex pattern.

```
TO PATTERN5
   CLEAR
   LINES 6Ø  128  1Ø
   HOME
END
```

```
TO LINES :LENGTH :X :STEP
   IF :LENGTH = 0 (STOP)
   SX :X SY 36 SH 0
   FD :LENGTH RT 90 FD :LENGTH
   LINES (:LENGTH - :STEP)
         (:X + :STEP) :STEP
END
```



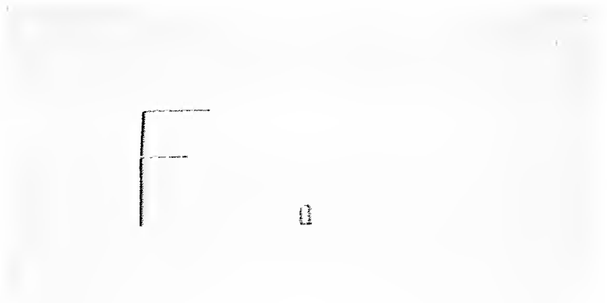
The starting points for the pattern are picked so as to center the pattern on the home position. Thus because home is at 128,96 the starting point for the first line is at 128,36 which is 60 units below home. We've chosen to orient the pattern so that the child can begin drawing without turning the turtle.

At some point the child will need practice in learning letters and numbers. Part of learning to recognize them is to look at them very carefully, and this can be encouraged by use of DOODLE mode activities. The child will probably want to use the letters later to write simple words, so we'll save the procedures they make. The first tasks could be simply copying from a model. Because most people identify computers with mathematics, here we'll counter that tendency by using letters for examples. We'll begin with the letter F. We need a procedure to draw the model.

```
TO DRAW-F
   CLEAR
   SX 50 SY 146 RT 180
   FD 50
   SX 50 SY 146 LT 90 FD 30
   SX 50 SY 126 FD 20
   HOME
END
```

This will draw a large capital F, as you can see by running the procedure. However, it will draw the F so quickly that it gives the child no hint as to the order in which the lines should be drawn. The order can be indicated in several ways. Color can be used (draw the red part, then draw the blue part). We can put delays between the strokes to make the sequence on the example visible. We'll use both techniques.

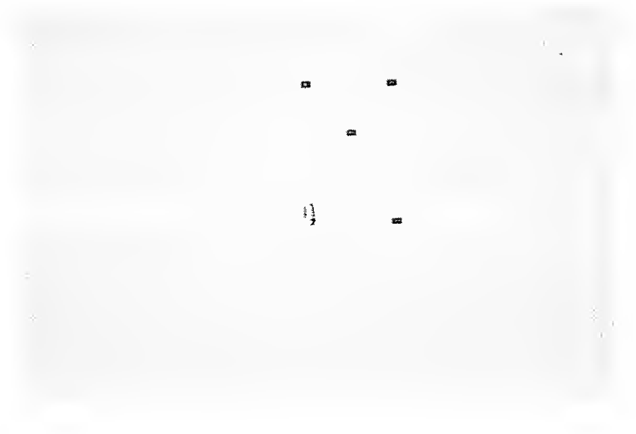
```
TO DRAW-F
   CLEAR
   SX 50  SY 146  RT 180
   PC 1  FD 50
   WAIT 6
   SX 50  SY 146  LT 90
   PC 2  FD 30
   WAIT 6
   SX 50  SY 126  FD 20
   WAIT 6
   HOME
END
TO WAIT :T
   REPEAT :T (REPEAT 820 ())
END
```



Notice that the procedure **WAIT** does nothing but count. The number on the inner **REPEAT** is picked so that the number **:T** is approximately the number of seconds that are used up counting. Be sure that the child's procedure is named **F** so that there is a simple correspondence between the name and the drawing. If you still have the OK Set in memory, you'll have to delete **F** from that set.

Once the child is familiar with the shape of the letter, or of several letters, you can let them try making letters by connecting dots. Here the procedure must draw the dots, preferably starting at the home position.

```
TO DOTM
   CLEAR
   DOT  FD 60  DOT  RT 135
   FD 30  LT 45  DOT
   LT 45  FD 30  RT 45
   DOT  RT 90  FD 60  DOT
   HOME
END
```

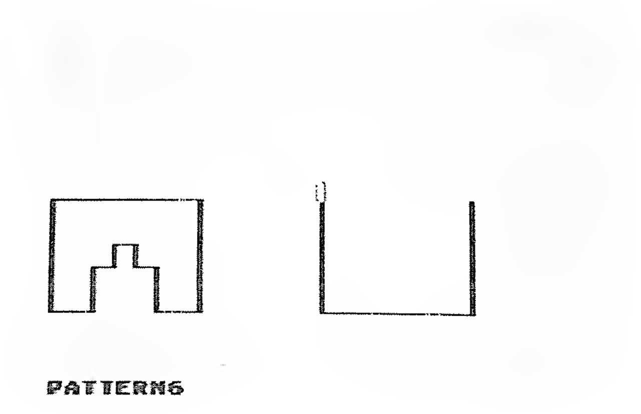


In this procedure we've been careful to always have the turtle pointing in the horizontal direction to keep the spacing of the dots perfectly regular. That may not be essential.

This dot-to-dot exercise works best for those letters and numbers where the pen never need be raised. Most letters require that the pen be raised. The dot pattern for these may be a bit of a puzzle, perhaps a worthwhile challenge. If that is too difficult, color coding the dots into two or three sets or adding intermediate dots may help.

The ability to visualize how things will appear in other positions may be worth developing. The idea here is to give a figure in one position and to ask the child to doodle it in another position. We are going to reuse the child's procedure for final comparison, so here we use DOODLE mode. One task is to ask the child to complete a partially drawn figure, but in another position.

```
TO PATTERN6
   RT 180
   SX 70  FD 50  RT 90  FD 20  RT 90
   FD 20  LT 90  FD 10  RT 90  FD 10
   LT 90  FD 10  LT 90  FD 10  RT 90
   FD 10  LT 90  FD 20  RT 90  FD 20
   RT 90  FD 50  RT 90  FD 70
   SX 198 SY 96  SH 180
   FD 50  RT 90  FD 70  RT 90  FD 50
END
```



Remember that in entering DOODLE mode it will be necessary to name the procedure that the child is creating. The comparison of the two figures can be made nicely. Let's assume that the child's procedure is named **ZZ**. After **ZZ** is completed, enter RUN mode and do the following.

```
RT 180
PATTERN6
SX 70
SY 146
PC 2
ZZ
```

This will rotate the original figure and draw the child's figure over the rotated original in another color. The result will be even more satisfying when the child is drawing the whole figure in the new position. Of course the above set of instructions could be combined into another procedure so as to speed the comparison.

The DOODLE mode projects can become quite complex. For example, a long DOODLE mode project could be teaching the turtle to write in handwriting. The key is to name each procedure for drawing a cursive letter with that single letter as the name. Thus, cursive a should be given the procedure name **A**, cursive b the procedure name **B**, etc. Then in RUN mode every time a letter is typed (followed by **ENTER**) the cursive letter will be drawn. Or one could define a procedure with a word spelled out (spaces between the letters) and the result would be the word in cursive letters.

```
TO CURSIVE
   C A T
END
```

will write *cat* in cursive if the procedures **C**, **A**, and **T** are correctly defined. To make it all work smoothly, the turtle will have to end up in the right position after each letter.

If that is not enough of a challenge, then how about making the computer draw letters as they appear in a manual on calligraphy? No doubt you will have to make them a bit bigger to get the desired effect. The only limitation is that you can't have both upper and lower case letters at once. There are limits, even with Color LOGO!

This chapter has given some idea of what can be done with the OK Set and DOODLE mode. At this point there are several ways you might continue with a child. One is to continue with DOODLE mode giving them even more challenging tasks or encouraging them to create useful procedures which you help them use in RUN mode. If they are reasonably good with the keyboard, they may not need much help. An alternative is to teach them how to extend the OK Set by adding procedures they create in DOODLE mode (they can make their own key labels and attach them as they name the procedures). For example, you could suggest that they draw a truck, a house, a tree, and a person in DOODLE mode, and then let them draw street scenes using these additions to the OK Set. Of course they'll soon want to add color. To do that they'll have to learn how to insert **PC** commands into their procedures, and before long they'll be typing and editing in the standard way.

14. MULTIPLE TURTLES

So far we have been drawing on the screen with a single turtle. Complex figures were made by drawing one piece at a time, first one piece completely, then a second piece, etc., until the drawing was complete. In Color LOGO there is another way to produce a complex drawing. We can draw all the parts at once using several turtles.

Multiple turtles provide many extra possibilities. Games are one obvious application. It is much easier to program games if each player is assigned a turtle which maintains its position until that player's next turn. This is especially true in Color LOGO as we often do not know the position of the turtle in absolute coordinates, in which case it is hard to store a return point. We shall see that it is often possible to simplify the programming by giving individual turtles different subtasks in a game. We can make drawings by assigning turtles to the tasks of drawing individual pieces of the whole. In this way the drawing will seem to evolve instead of appearing piecemeal. At a more serious and abstract level, we can use the multiple turtles to illustrate the process of multiprogramming.

Let's begin by entering a few procedures for the turtles to run. Clear the memory and enter

```
TO BOX :SIDE :X :Y
   SX :X SY :Y
   REPEAT 4 (FD :SIDE RT 90)
END

TO CIRCLE :SIDE :X :Y
   SX :X SY :Y
   REPEAT 20 (FD :SIDE RT 18)
END
```

Run each of these to verify that they are entered correctly.

We create new turtles by means of the HATCH command. The form of the command is

```
HATCH turtle number procedure for the turtle
```

or to give a specific example

```
HATCH 1 BOX 50 30 60
```

Here the meaning of **HATCH** is obvious. The following number, here 1, is the name or label of the turtle. Turtles can be labeled with any number between 1 and 254. (Turtle 0 is the master turtle—always present, even if hidden by an **HT** command—which we have been using exclusively up to now.) **BOX** is the name of the procedure which we are telling turtle 1 to run. The numbers following **BOX** are, as usual, the values to be fed into the local variables within **BOX**.

Next we try out a simple multiple turtle program. Enter the following

```
TO TEST
   HATCH 1  BOX 5Ø 3Ø 6Ø
   HATCH 2  BOX 4Ø 18Ø 9Ø
   HATCH 3  BOX 6Ø 1ØØ 9Ø
END
```

Notice that each turtle has its own procedure and its own set of values for the variables. Of course several turtles can be using the same procedure, but each still has its own current set of values for the variables. When you run it you may get less than you expected. Why does the program stop before drawing the boxes? Notice that there are four turtles on the screen: the three you created with the **HATCH** commands and, as always, the master turtle. When there are multiple turtles, Color LOGO gives each turtle a turn in sequence. A turn is a single turtle command or a logical operation in a control statement. (A *control statement* is a statement which controls the sequence of operations in the procedure, for example: "**IF**" or "**REPEAT**.") Therefore having created turtle 1, Color LOGO gives turtle 1 a turn and creates turtle 2, then it gives turtles 1 and 2 each a turn and creates turtle 3. Next it gives turtles 1, 2, and 3 each a turn and encounters the **END** instruction. Because turtle Ø is always there, turtle Ø is now waiting to find out what to do. We have not given turtle Ø anything else to do, so it is waiting for a command from the keyboard. If we press $\boxed{\text{ENTER}}$ (a command for turtle Ø to do nothing), then all the other turtles get another turn. Try it. Remember this as a way to single-step through procedures to find errors.

Of course we do not always want to have to sit at the keyboard pressing $\boxed{\text{ENTER}}$. We can get the whole thing to work as planned if we give turtle Ø some procedure to run as well. Try

```
TO TEST
   HATCH 1  BOX 5Ø 3Ø 6Ø
   HATCH 2  BOX 4Ø 18Ø 4Ø
   HATCH 3  BOX 6Ø 1ØØ 2Ø
   BOX 2Ø 15Ø 12Ø
END
```

The last call of **BOX** has no **HATCH** preceding, so it is addressed to turtle Ø. That's more like it! If you want to see in a bit more detail what is actually happening you might want to slow down the speed. You can slow any procedure by inserting a **SLOW** command.

```
TO TEST
   SLOW 3Ø
   HATCH 1  BOX 5Ø 3Ø 6Ø
   HATCH 2  BOX 4Ø 18Ø 4Ø
   HATCH 3  BOX 6Ø 1ØØ 2Ø
   BOX 2Ø 15Ø 12Ø
END
```

The number after **SLOW** tells the computer how much to slow down. The number must be between Ø and 127. Zero is full speed and 127 is the slowest speed. The **SLOW** command sets a speed for all procedures which will remain unchanged until reset with another **SLOW** command or until RUN mode is exited and then reentered.

Before we leave this example, notice that at the completion of each turtle's procedure the turtle disappears so that at the end only turtle Ø remains.

Of course we can use many procedures for the various turtles. Try

```
TO TEST2
  HATCH 1  BOX 5Ø 3Ø 6Ø
  HATCH 2  BOX 4Ø 18Ø 9Ø
  HATCH 3  BOX 6Ø 1ØØ 2Ø
  HATCH 4  CIRCLE 3 3Ø 14Ø
  HATCH 5  CIRCLE 4 18Ø 12Ø
  CIRCLE 5 9Ø 9Ø
END
```

This procedure can be used to point out one potentially troublesome point. What if we altered the procedure by making the procedure for turtle Ø **BOX** (say **BOX 8Ø 9Ø 9Ø**)? If you try this, you will find that the circles are not completed and the two turtles drawing the circles remain on the screen. This is because turtle Ø runs out of commands before the others are finished. To avoid the problem, always put the procedure for turtle Ø last, and assign turtle Ø the most complex procedure.

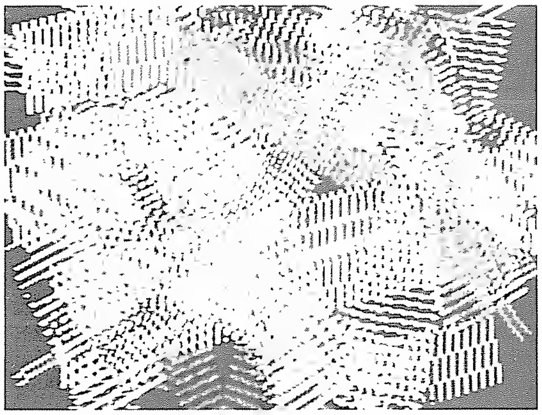
Another solution to the problem mentioned above is contained in the procedure **ABSTRACT**.

```
TO ABSTRACT
  CLEAR COLORSET 1
  RT 25
  HATCH 1 PATH 1 4 3Ø
  RT 43
  HATCH 2 PATH 2 4 2Ø
  RT 67
  HATCH 3 PATH 3 4 4Ø
  RT 1Ø5
  HATCH 4 PATH Ø 4 1Ø
  VANISH
END
```

Notice that turtle Ø is turned between each **HATCH**. The initial position and heading of each new turtle is the same as that of the parent turtle (the turtle which hatches the new one). In this example turtle Ø is the parent, so each new turtle will have the position and heading of turtle Ø at the time of HATCHing. After the four new turtles are created, then turtle Ø is given the **VANISH** command. The **VANISH** command tells a turtle to go out of existence. Once turtle Ø is out of existence, it no longer gets a turn, and it cannot bring the procedure to a halt by running out of commands.

Of course this procedure needs **PATH** to function.

```
TO PATH :COLOR :I :L
  HT  PC :COLOR
  WHILE 1
     (FD :L  RT 90  PU  FD :I
      RT 90  PD  FD :L
      LT 90  PU  FD :I  LT 90  PD
  IF NEAR 255 > 150
           (RT 108)
  )
END
```



PATH contains some new ideas which we should explain. The first is the **WHILE** statement. The **WHILE** is somewhat like a **REPEAT**, but with a condition. The most common use is to repeat while some condition is true (for example, **WHILE :X < 3**). The computer evaluates the condition and returns the value zero if the condition is false or a non-zero value if the condition is true. Here we want it to repeat forever, so we assign the condition the value 1 which always is interpreted as true. The parentheses following the **WHILE** enclose the commands which are to be repeated. The other new idea is the use of the **NEAR** function. The **NEAR** function returns an indication of the distance from the current turtle to the designated turtle. Actually, what you get is the total of the steps in the x direction and in the y direction to the designated turtle. In **PATH** the statement is

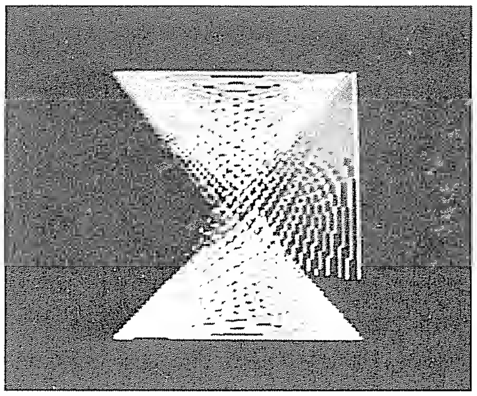
```
IF NEAR 255 > 150 (RT 108)
```

The current turtle (remember 1, 2, 3, or 4) is asking the distance to turtle 255. But turtle 255 does not exist. When you request the distance to a non-existent turtle, you get the distance to the home position. Therefore this statement says, if the current turtle is more than 150 steps away from home, then turn right 108 degrees.

After such a long explanation, we should get a program which runs a long time. **ABSTRACT** will surely fit the bill; it will run until you hit the **BREAK** key or until there is a power failure.

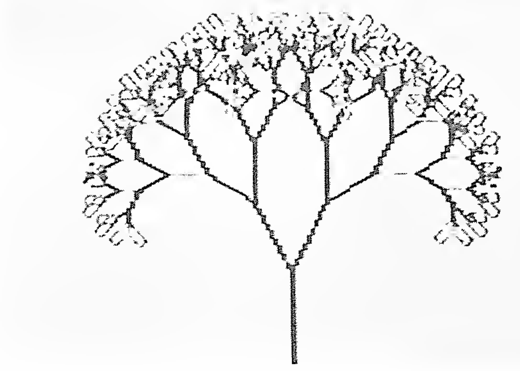
Perhaps you'd like a different design.

```
TO MIXIT
COLORSET 1  BG 0
HATCH 1 SWEEP 1 3  60  30    0
HATCH 2 SWEEP 2 3  60 160  90
HATCH 3 SWEEP 3 3 190 160 180
HATCH 4 SWEEP 2 3 190  30 270
VANISH
END

TO SWEEP :COL :INT :X :Y :H
   REPEAT 12
        (HT  PC  :COL
         SX  :X  SY :Y  SH :H
         REPEAT 92/:INT
             (PD  FD 100  PU  BK 100
              RT  :INT
             )
        MAKE  :COL  :COL+1
        )
END
```



There is no question that the turtles slow down as the number of turtles on the screen increases. After all, more is going on. Thus far we haven't had so many that the slowing is that noticeable. But how about a program which generates a lot of turtles? One interesting test is to return to a recursion program and implement it using multiple turtles. **TREE** is an ideal example. Try the following.

```
TO TREE :S
   IF ME=0 (CLEAR SETY 0)
   IF :S>6
      (FD :S  LT 30
       HATCH 1 TREE (3*:S/4)
       RT 60
       HATCH 2 TREE (3*:S/4)
       VANISH)
END
```



Again we've introduced a new idea with this procedure, here the **ME** function. The **ME** function returns the identification number of the current turtle. The statement

```
IF ME=Ø  (CLEAR SETY Ø)
```

says if the current turtle is turtle Ø, then clear the screen and move. Because turtle Ø is subsequently told to **VANISH**, this will happen only once. This procedure recursively hatches new turtles, all named either 1 or 2. Because the recursive calls keep levels distinct, this is satisfactory, but functions like **NEAR** would give unpredictable results because the various turtles are not uniquely named.

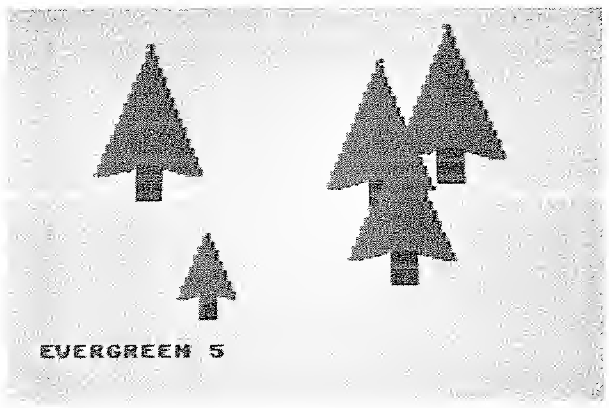
There are a number of interesting things that can be tried with this procedure. One is to compare it in speed with the earlier version of **TREE**. In one case you have all the backing up necessary for a pure recursive program, and in the other you have the overhead necessary to keep track of all the turtles. To make the comparison meaningful you'll have to make the two versions draw the same size tree, but by now that is easy. The comparison may give some idea of why multiprogramming is worth learning about. You can speed the multiple-turtle version by reducing the number of times the turtle has to be drawn. Simply insert a **HT** command as the first command in the procedure. One other change converts the tree into full blossom. Try

```
TO TREE :S
   IF ME=Ø  (CLEAR SY Ø)
   IF :S>6
      (FD  :S  LT 3Ø
       HATCH 1 TREE (3*:S/4)
       RT 6Ø
       HATCH 2 TREE (3*:S/4)
       VANISH)
      ELSE (REPEAT 5ØØ () )
END
```

The addition is the **ELSE** statement. The **ELSE** is a partner of the **IF** statement. The combination says if the current value of **:S** is greater than 6 then obey the commands in the following set of parentheses (from **FD :S** through **VANISH**, but if **:S** is not greater than 6 then obey the commands in the parentheses following **ELSE**. The commands following **ELSE** simply delay the completion of the procedure so that we can see the tree with a turtle at the end of each branch.

Trees are so easy to draw with multiple turtles that we may as well draw a complete forest. In fact we'll look at two forests, one a deciduous forest in winter and the other an evergreen forest in whatever season you like.

```
TO FIR1  :N  :X  :Y
  HT  SX :X  SY :Y  PC 0
  BK :N/2  RT 90  FD :N/4
  LT 90  FD 6+:N/2  RT 90
  FIR 11  :N  :X
END

TO FIR11  :N  :X
  PC 1  RT 15  FD :N
  LT 129  FD 3*:N
  WHILE XLOC ME>:X  (FD 2)
END

TO FIR2  :N  :X  :Y
  HT  SX :X  SY :Y  PC 0
  BK :N/2  LT 90  FD :N/4
  RT 90  FD 6+:N/2  LT 90
  FIR22  :N  :X
END

TO FIR22  :N  :X
  PC 1  LT 15  FD :N
  RT 129  FD 3*:N
  WHILE XLOC ME<:X  (FD 2)
END

TO FIR  :N  :X  :Y  :T
  HT
  HATCH :T  FIR1  :N  :X  :Y
  HATCH :T+1  FIR2  :N  :X  :Y
  IF :N>20  (STOP)
  FIR (:N+1)  :X  :Y  :T
END

TO EVERGREEN  :TREES
  HT
  WHILE :TREES > 0 (
      MAKE :X RANDOM 200 + 20
      MAKE :Y RANDOM 100 + 30
      MAKE :T :TREES*3
      HATCH :T FIR 2 :X :Y :T
      REPEAT 30 ()
      MAKE :TREES :TREES-1)
  VANISH
END
```



Try running this set of procedures, first with one tree and then with several. With some TV sets you may be able to get green tops and brown trunks, so try playing around. If not, you can always claim that they're intended to be blue spruce. There are a couple of new ideas in the last two procedures. In **FIR11** we have used the **XLOC** function. This returns the X screen coordinate of the designated turtle. Here **:X** is the starting point for the right half of the tree. When **XLOC** has returned to the starting point, the procedure is finished. In **FIR** notice the use of the variable **:T** to indicate the turtle number. In **EVERGREEN** we have introduced the **RANDOM** function. **RANDOM** produces a random number between Ø and the argument–1. For example

```
RANDOM 2ØØ + 2Ø
```

adds 2Ø to a random number between Ø and 199. The result must be a number between 2Ø and 219. We do this to keep the trees away from the edge where the wrap-around will give some rather lopsided trees. Also note the use of **WHILE** in combination with the

```
MAKE :TREES :TREES–1
```

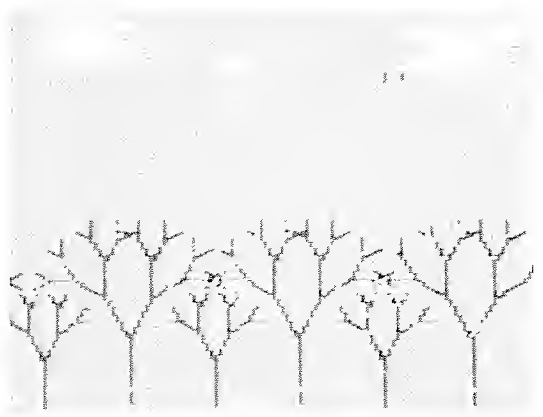
This gives a number which is one less every time through the **WHILE** loop. **:TREES** is used to vary the turtle numbers for each tree drawn.

The deciduous forest uses the **TREE** procedure we've already seen.* To that we must add

```
TO FOREST
   BACKGROUND 1
   SX 236
   REPEAT 3 (SY 1Ø
        SX XLOC ME + 4Ø
        HATCH 1 TREE 2Ø
        SX XLOC ME + 4Ø
        HATCH 2 TREE 3Ø)
   CLOUDS
END
```

*(Use the tree procedure from page 75.)

```
TO CLOUD :SIZE :X
   SETHEADING 90
   REPEAT (:SIZE/6)
        (MAKE :X RANDOM (:SIZE/2)
         PU  FD :X/2  PD
         FD :SIZE-:X  PU
         BK :SIZE-:X/2
         SY YLOC ME-2)
END

TO CLOUDS
   PC 2  SX 10  SY 180
   CLOUD 60
   SX 100  SY 164
   CLOUD 30
   SX 190  SY 176
   CLOUD 65
END
```



Again in this example we have created many turtles with the same numbers by hatching them recursively. The two multiple tree drawings show the two ways in which multiple turtles can be created. It makes no difference which way you do it unless you are going to refer to the turtle by number. In that case, each turtle must have a unique number.

Notice the statement

```
SX XLOC ME + 40
```

This has the meaning

```
SX (XLOC ME) + 40
```

not

```
SX XLOC (ME + 40).
```

15. NEW SHAPES FOR TURTLES

All turtles are created equal; at least they all look the same. In the examples we have seen so far that didn't matter, but often we want the different turtles to look different. For instance, it would be impossible to play many games if all the pieces or players looked the same. So Color LOGO includes a way to change the shape of individual turtles. As we shall see, this gives us a bonus: a way to do simple animation.

The shape of the turtle is changed by means of the **SHAPE** statement. Following the **SHAPE** is a list of turtle shape commands. Turtle shapes are drawn using a very limited set of turtle graphic commands, basically forward and back a single step, right or left by 45 degrees, and penup and pendown. The commands in a **SHAPE** statement have absolutely no effect on the turtle position, heading, or pen state. The symbols used for these commands are listed in the following table.

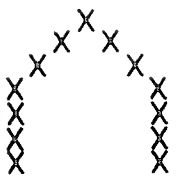
TURTLE SHAPE COMMAND	EFFECT
F	Step forward one dot. If the pen is down, complement (reverse the color of) the dot.
B	Step backward one dot. If the pen is down, complement the dot.
R	Turn right 45 degrees.
L	Turn left 45 degrees.
U	Pick up the turtle shape pen. This pen is always down at the start of a **SHAPE** command. The turtle shape pen is completely independent of the standard turtle pen; **PU** and **PD** commands have no effect on the turtle shape pen, and **U** and **D** have no effect on the turtle pen.
D	Put the turtle shape pen down. If the turtle shape pen was up, the putting it down will cause the current dot to be complemented.

Notice that because the only forward or back possible is one dot, then the only turns which make sense are 45 degree turns.

This will be clearer if we give an example. Let us assume that the current orientation of the turtle is heading straight up. Then the command

```
SHAPE URRFFFLLDFFFFL-
FFFLLFFFLFFFF
```

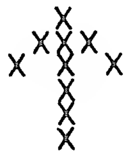
will draw the following turtle shape.



Notice that the turtle shape commands can extend over more than a single line. Multiple lines must be connected with a hyphen and must start in column 1. This in turn means that there is no limit on the size or complexity of a turtle shape. However, the turtle shape must be redrawn every time the turtle moves, so the larger and more complex the turtle shape, the slower the system will run.

It is fairly difficult to create desired turtle shapes by trial and error at the keyboard. It is especially difficult to locate an error in the middle of a string of turtle shape commands. We have found the following to be effective ways to proceed. First design the turtle shape on a piece of graph or engineering paper. The possibility of rotating the paper as you enter the shape may save you from getting a stiff neck trying to play turtle at the keyboard. Once the shape is designed on the graph paper, there are two methods which we use. If the turtle shape is a simple one, enter the shape in DOODLE mode. Remember that the keys 3, 4, 5, 6, and 7 correspond exactly to the turtle shape commands **U**, **D**, **R**, **L**, and **F**. Only **B** is missing, and while B is very useful, it is not essential. Use of DOODLE mode continually shows you the current heading, which is a big help. However, the turtle drawn in DOODLE mode will not look exactly like the final turtle for several reasons. Lines drawn in DOODLE mode are two dots wide, but lines drawn as turtle shapes are one dot wide. Also, lines which cross, complement when they are parts of turtle shapes, but they do not complement in DOODLE mode. However, you will get to see the shape in about the final form and of exactly the final size while drawing it. Once you have the shape completed in DOODLE mode you can enter EDIT mode and convert the procedure into one to draw a new shape. Simply insert the **SHAPE** command before the command list and convert each **3** to **U**, each **4** to **D**, each **5** to **R**, each **6** to **L**, and each **7** to **F**, all by overtyping.

Let's begin with a very simple example. We want the turtle to appear as an arrow. On graph paper we draw the dot pattern.



The actual turtle position is to be at the tail end of the arrows. Enter DOODLE mode, naming the procedure **NEW**, and draw the figure. The keystrokes are **777777766677555537755747**. You may be surprised at the small size of the turtle, but you can always draw a new one after you've learned the technique. Now enter EDIT mode and look at **NEW**. Insert **SHAPE** before the list of symbols and replace the DOODLE symbols with the appropriate turtle shape commands. Don't forget to include the hyphen at the end of the first line of commands. At this point your procedure should be

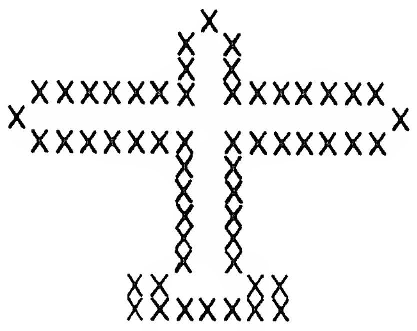
```
TO NEW
SHAPE FFFFFFFLLLFFRRRR-
UFFRRFDF
END
```

To see how your shape looks, run **NEW** and then enter commands like **FD 20** and **RT 90**. Remember that the turtle can be drawn in eight positions. Be sure to try the diagonal positions (for example, **RT 45**) because there will be some change in shape as the turtle rotates to these positions. To see why that is so, return to the graph paper and follow your turtle instructions beginning along a diagonal. It is a good idea to do this on graph paper before going to the computer, as your turtle shapes might come apart upon rotation if they are drawn in the wrong sequence. To show you that it can happen, try the following. We could have drawn essentially the same shape by the steps

```
FFFFFRRUFFDLLLFFLLFF
```

in the vertical or horizontal positions, but in the diagonal positions this pattern comes apart, as you can see if you follow the instructions on graph paper.

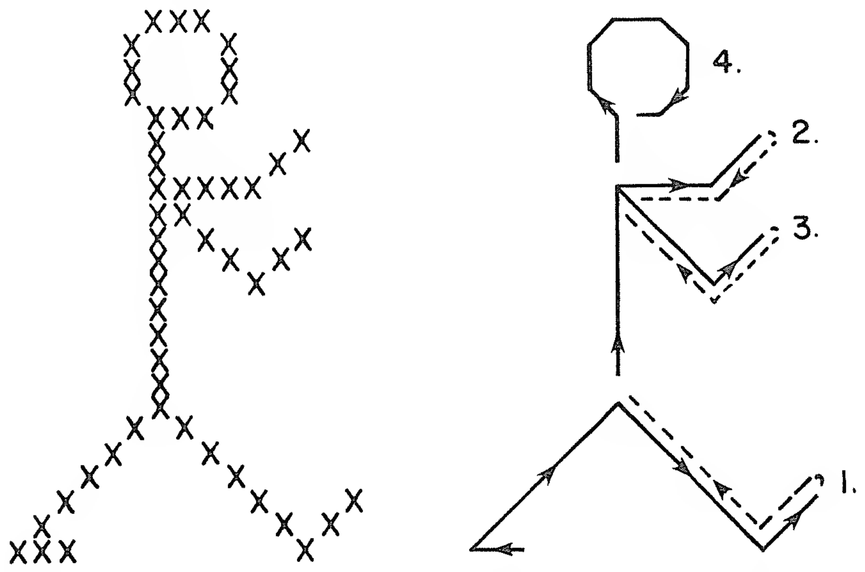
Now we move to a bit more complex example, an outline of a plane. The dot pattern is



and the procedure created in DOODLE mode and translated in EDIT mode is

```
TO PLANE
   SHAPE RRFFFLLFLLFRFR-
   FFFFFFRRFFFFFFF-
   LFLLFLFFFFFFFRRF-
   FLFLLFLFFRRFFFF-
   FFFLFLLFLFFFFFF-
   FRRFFFFFFRFRFLL-
   FLLFF
   END
```

When we go to more complex shapes we prefer to work with paper rather than with DOODLE mode. There are several reasons for this. One is that the **B** command is very useful. The other is that we must be very careful if we are to avoid problems when the turtle shape is turned to the 45 degree positions. As is our custom we will illustrate with an example. We want to use the following stick figure as a turtle shape.



We must choose our pathway through the figure carefully. The problem points are points where lines meet. To avoid problems we avoid shortcuts and return to junction points by backtracking exactly. The pathway we take is indicated on the second figure. The dotted lines indicate backtracking with the pen raised. Remember, if we did not raise the pen when backtracking, then the lines would be complemented a second time (that is, erased). The only place where we don't backtrack is on the head. If a closed figure is symmetric, then it will stay closed when rotated.

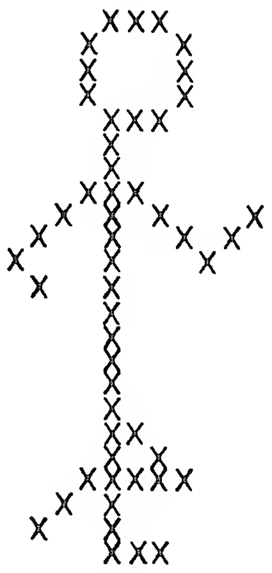
The other point to note carefully is when to raise and lower the pen. A dot will be complemented if the pen is down when we move into the dot or if the pen is lowered while we are in the dot. Notice that this means that if we draw a line and then cross that line with the pen down, the crossing dot will be erased.

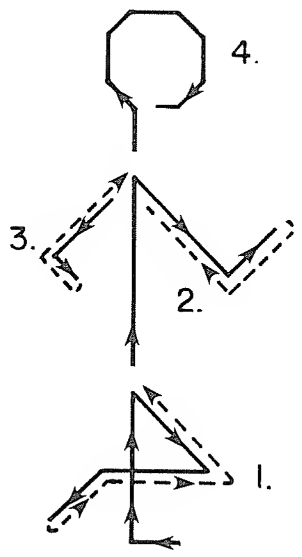
Following the pathway indicated gives the following procedure. As stated earlier, we would work this out on paper and enter it directly in EDIT mode so that we could use the B command.

```
TO ONE
  SHAPE LLULLFFFFDFFRRRFFFFFF-
RRFFFFFFFLLFFUBBLLFFFFFFRFD-
FFFFFFFFRRFFFFLFFUBBRBBBB-
RFDFFFLLFFUBBLLFFFFRFD-
FFLFRFFRFRFFRFRFFRFRF
END
```

Try this out by running **ONE**. Try rotating it to other positions. Notice that when you turn it far enough it is upside down. You may not want turtles that do strange things like that for some purposes. You can sometimes avoid it if it is a problem by shifting your point of view. For example you might decide that it would be nice to have a turtle which actually looked like a turtle. If you draw a side view of the turtle, then it will look strange with headings like 180 degrees. But if you draw a top view of the turtle, then it looks fine in any orientation.

The reason that we drew the stick figure is that we want to show you how to use Color LOGO to do some very simple animation. We want to have a figure that will walk. We'll need another position for the stick figure, so we define another turtle shape. The process is the same as before.

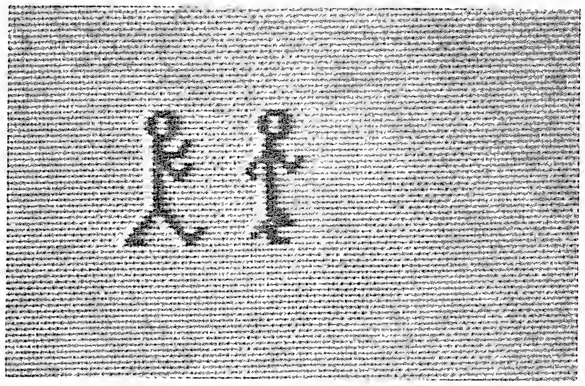




Translating the indicated path into a procedure gives

```
TO TWO
   SHAPE LLURRFFDBBLLFFFFFFRRR-
FFFRRRFFFFLFFUBBRBBBBRFFFRFD-
FFFFFFFFFRRRFFFFLLFF-
UBBLLFFFFLLFDFFFLLFU-
BLLFFFFLFDFLFRFFRFR-
FFRFRFFRFRF
END
```

Now that we have the shapes, we can have some fun. First let's make them walk.



```
TO WALK
   HT  PU  SX 100  RT 90
   REPEAT 100  (ONE  ST  WAIT 100
        HT  FD 6  TWO  ST
        WAIT 100  HT  FD 6)
END

TO WAIT :T
   REPEAT :T ()
END
```

Notice that in this case we want the turtle shape to be drawn at right angles to the turtle motion. That is taken care of in the **SHAPE** statement. Notice also that we have to slow down the process by including the **WAIT** instruction. Otherwise it runs so fast that we have trouble seeing the shape. Try other values for **:T** to vary the speed. We can make the figure climb; just enter **LT 15** before running **WALK** again. We can even make the figure walk in a circle.

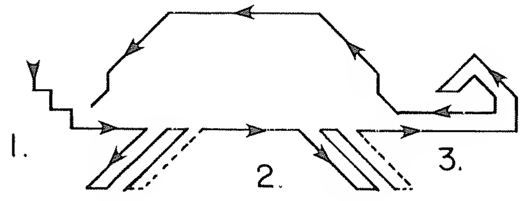
```
TO WALK-AROUND
   HT  PU  SX 100  RT 90
   REPEAT 100  (ONE  ST  WAIT 100
             HT  RT 15  FD 6
             TWO  ST  WAIT 100
             HT  RT 15  FD 6)
END
```

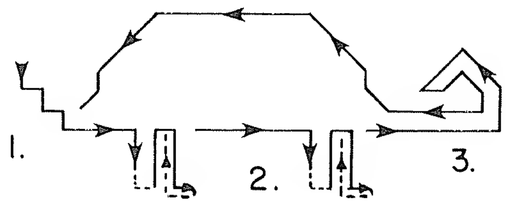
You may prefer the motion you get with a different control procedure. Try this as an alternative to **WALK**

```
TO WALK1
   HT  PU  SX 100  RT 90
   REPEAT 100
     (HATCH 1 WALKA
      REPEAT 8 ()
      HT  FD 6
      HATCH 1 WALKB
      REPEAT 8 ()
      HT  FD 6
     )
END

TO WALKA
   HT  ONE  ST
   REPEAT 10 ()
END

TO WALKB
   HT  TWO  ST
   REPEAT 10 ()
END
```

The trick here is to get the delays (that is, the **REPEATS** with empty parentheses) synchronized. The delays in **WALK1** must match with the delays in **WALKA** and **WALKB**. If there is a mismatch one direction the two figures will appear together, and if there is a mismatch the other way the motion will be unnecessarily jerky.

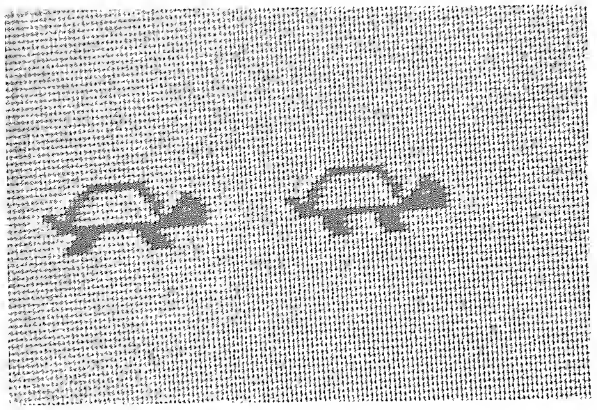
After all this talk about turtles, we feel an obligation to actually draw something which looks a bit like a turtle. As our last example we give a HERD of turtles.





```
TO TURTLE1
   SHAPE LL-
BRRFRRFLLFRRFLLFFFFLBBBRFL-
FFFRFLBBBUFFFRFDFFFFFRFFF-
LFRBBBLFRFFFUBBBLFDFFFFFFF-
LLFFLFFLLFFRBLBLLFRFRRFFFFF-
RFRFLFFFLFFFFFFFFLFFFLFRF
END

TO TURTLE2
   SHAPE LL-
BRRFRRFLLFRRFLLFFFFRRFFUBB-
LLFDRRFFFLLFFUBLLFDFFRRFFFFFFF-
RRFFUFLLFDLLFFFRRFRRFFFLLFUB-
LLFFFRRFDFFFFFFF-
LLFFLFFLLFFRBLBLLFRFRRFFFFF-
RFRFLFFFLFFFFFFFFLFFFLFRF
END


TO CRAWL :T :X :Y
   HT  PU  SX :X  SY :Y
   RT 90
   REPEAT 100 (
        HATCH :T+1 T1
        REPEAT 8 ()
        HT  FD 2
        HATCH 1 T2
        REPEAT 8 ()
        HT  FD 2
        IF XLOC ME > 230 (VANISH)
   )
END
```

```
TO T1
   HT  TURTLE1
   ST
   REPEAT 10 ()
END

TO T2
   HT  TURTLE2
   ST
   REPEAT 10 ()
END

TO HERD
   CLEAR  HT
   MAKE :I 0  MAKE :T 1
   REPEAT 20 (
        IF :I<10  (MAKE :I :I+1)
        MAKE :J 1
        WHILE :J<:I
        (HATCH :T CRAWL :T 0 (:J*18)
         MAKE :T :T+2
         MAKE :J :J+1)
         REPEAT 900 ()
   )
END
```



It is clear that while much is possible with the turtle shapes, Color LOGO is not likely to become a tool for the generation of Saturday morning TV shows. It was never intended that it should be so. It is a tool that will allow the child to produce results which can be immensely satisfying to the creator.

16. GAMES

One of the most popular applications of computers is gaming. Color LOGO can be used to create a great variety of games. In this chapter we will give two examples of turtle games. These are included not as competitors for the local video arcade, but as illustrations of some very useful techniques for communication between turtles.

Before getting into the details of the simple game we're going to use, we want to point out a few things which may be obvious. Most of the popular video-arcade and computer games rely very heavily on speed. Things happen which force the player to react faster and faster until finally they fail. You've already gotten some feel for the speed at which animation runs in Color LOGO; it's not going to be fast enough to create shoot-em-up space games that will hold interest for long. However, it does have capabilities such that the user can create rather than just play such games. If you want to create games which will also be challenging to play using Color LOGO, then you might try to think of games where coordination of several moving objects is the challenge (thus lower speed is no limitation) or games where there is sufficient strategy that the player must think while playing.

Our first sample game is called **CATCHEM**. There are two players (or one two-handed player) who manipulate objects on the screen by pressing keys on the keyboard. The object is for the chaser to catch the runner. When the chaser catches the runner, the scorekeeper changes the score, and a new chase starts. There is an advantage to using multiple turtles here, as we can assign each turtle one task. This simplifies the programming greatly; for example, we do not have to move a cursor to the scoreboard to change the score and then return to make the next move.

The master procedure simply names the procedures and assigns the four tasks to four turtles. We use turtle Ø and three others.

```
TO CATCHEM
  CLEAR
  HATCH 1 GETKEYS
  HATCH 2 RUNNER 2Ø
  HATCH 3 CHASER
  SCOREKEEPER Ø
END
```

The names of the procedures for the four turtles are pretty descriptive. **RUNNER** controls the runner and **CHASER** controls the chaser. **SCOREKEEPER** keeps the score. **GETKEYS** reads input from the keyboard. Of course the various turtles need to communicate, and that is the main new idea we will illustrate in this example.

Let's begin with the keyboard.

```
TO GETKEYS :X
  HT
  WHILE 1 (MAKE :X KEY
    IF :X = 'S (SEND 2 1)
    IF :X = 'D (SEND 2 45)
    IF :X = 'A (SEND 2 315)
    IF :X = 'K (SEND 3 1)
    IF :X = 'L (SEND 3 45)
    IF :X = 'J (SEND 3 315)
  )
END
```

First we see a trick we used before: the use of **WHILE 1** as an effective **REPEAT FOREVER**. The second new item is the **KEY** function. The **KEY** function looks at the keyboard to see if any key has been pressed. If no key has been pressed, then **KEY** returns the value Ø. Thus, if at the time turtle 1 is executing the statement

```
MAKE :X KEY
```

no key is depressed, then the variable **:X** is given the value Ø. If on the other hand a key is depressed, then the variable **:X** is given the ASCII value of the key. So the **KEY** function returns either the ASCII value of the key depressed or Ø if no key is depressed. The ASCII value is a number assigned to each key on the keyboard according to an industry-wide convention. In this procedure we do not have to know what the particular number is because the literal (for example, **'S**) automatically computes the ASCII value as well.

The next task for this procedure is to recognize which key has been depressed and to send a message to the appropriate turtle. We have to decide what keys to use for what actions of the runner and the chaser. We decided on the following key assignments.

S	—	move runner forward
A	—	turn runner left
D	—	turn runner right
K	—	move chaser forward
J	—	turn chaser left
L	—	turn chaser right

So now we see what (if any) key was pressed. First look at the statement

```
IF :X = 'S (SEND 2 1)
```

The literal **'S** gives the ASCII value of the argument **S**. That is, the condition **:X = 'S** in combination with the previous **KEY** function checks to see whether the **S** key was depressed. If the **S** key was depressed, then the statement **SEND** is run.

The **SEND** statement sends a message to another turtle by leaving the message in a mailbox. The first number after the **SEND** is the address of the message. In the line we are analyzing the address is 2, so this message can be picked up from the mailbox only by turtle 2. The address can be an expression as well as a number. The second number after the **SEND** is the message. Here the message is the number 1; in general the message can be any number in the range covered by Color LOGO (-32768 to 32767) or an expression which gives a number in this range. To review,

```
SEND 2 1
```

leaves the message **1** in the mailbox for turtle **2**. Because the **S** key is to move the runner (turtle **2**), the message **1** must mean move. We'll see that in the procedure **RUNNER**.

Although we aren't going to use it in this example, there is a way to send a general message to the first turtle that picks up its mail. We just use the turtle address 255; then the next turtle that inquires will get the message. If we wanted to send an all points bulletin to all turtles, we could do so by setting a global variable (see Chapter 6).

The rest of **GETKEYS** is just more of the same. We check for each of the keys which control the runner and send a message to turtle 2 if one of them is depressed, and we do the same for the three keys which control turtle 3. Notice that the **WHILE 1** causes turtle 1 to continue to poll the keyboard forever. There are certain features of Color LOGO which make this part of the programming very simple. By assigning one turtle the task of watching the keyboard at all times we make sure that the two players have equal access to control; we are very unlikely to lose keystrokes while something else is happening, and provision for regular polling of the keyboard is handled automatically by the logic which handles multiple turtles.

Now let's turn our attention to **RUNNER**.

```
    TO RUNNER :X
      PU  SX :X
      SHAPE FFFFFFFFUBBBRRFD-
    FFUBBBDBBB
      WHILE 1  (MAKE :X  MAIL 1
          IF :X
            (IF :X = 1 (FD 8)
             ELSE (RT :X)
)
      )
    END
```

RUNNER sets a starting position for the runner, lifts the pen so that the runner leaves no tracks (which makes no difference in the chase, but keeps the screen clean), and draws a shape so that the runner will look different. We then enter another **WHILE 1**, which will run forever.

The runner turtle now checks its mailbox by using the **MAIL** function. The number following **MAIL** (the argument) is the number of the turtle that the runner turtle will accept mail from. Here turtle 2, the runner turtle, is asking for mail from turtle 1, the keyboard turtle. If there is no message, then **MAIL** returns the value Ø. The statement

```
IF :X
```

checks for the value of **:X**. If it is Ø, then the statements in parentheses are skipped. Since the parentheses enclose all the rest of the commands, a Ø causes the loop to start again. Thus the turtle just keeps checking its mail until it gets a message from turtle 1.

If we look back at **GETKEYS** we see that a message 1 meant to move. Therefore if **:X = 1** the runner is moved forward 8. If at this point the message is not 1, then it must be either 45 or 315. The runner is turned right by either amount (remember that **RT 315** is the same as **LT 45**). This completes the move, so the turtle goes back to checking its mail from turtle 1.

Before going further, look carefully at the arrangement of the two **IF** statements in **RUNNER**. Notice that the parentheses after the first **IF** enclose the second **IF** and the **ELSE**. This pairs the **ELSE** with the second **IF**. The meaning is: if **:X** is not Ø (the first **IF**), then do one or the other of the following; if **:X** is 1 move forward—otherwise turn.

The **CHASER** procedure is similar to **RUNNER**, but it includes the test for a successful catch.

```
TO CHASER :X
  WHILE 1
     (HOME  PU
   WHILE NEAR 2 > 12
        (MAKE  :X  MAIL 1
          IF :X
            (IF :X=1  (FD 16)
             ELSE  (RT :X)
            )
     )
   SEND Ø 1
   )
END
```

CHASER includes nested **WHILE** statements. The first one starts the chaser and runs forever. The inner one runs until a capture is made. The definition of a capture is that the value returned by the **NEAR** function is 12 or less. The portion of the procedure controlled by the condition **NEAR 2 > 12** is identical to that in **RUNNER**. Remember that the **NEAR** function returns the total number of X and Y steps from the current turtle to the designated turtle, here to turtle 2. Thus the inner part of the procedure says to continue checking mail and making moves as long as the runner is more than 12 steps away.

If the runner is not more than 12 steps away, then turtle 3, the chaser, sends a message (1) to the scorekeeper (turtle Ø). Having sent the message, the chaser returns to the home position and the chase begins again.

Now we turn to the procedure for the scorekeeper.

```
TO SCOREKEEPER :S
   HT  SX 200  SY 180
   WHILE 1
      (PRINT "  " PRINT :S
       WHILE  MAIL 255 = 0 ()
       MAKE :S  :S + 1
      )
END
```

Again there are several new ideas in this procedure. The first steps are to hide the turtle and to position it to keep the scoreboard. We set the initial score to 0 by the call of the procedure and again use a **WHILE 1**. The **PRINT** statement causes anything contained in quotes to be printed on the screen at the current turtle position. The turtle is not moved. The **PRINT** statement also can be used to print the current value of a variable in the same way. Both are used here. **MAIL 255** is a special version of the **MAIL** function. **MAIL 255** will accept messages from all other turtles. Here we could use **MAIL 3** just as well, since turtle 3 is the only one sending messages to the scorekeeper. The following line:

```
WHILE MAIL 255 = 0 ()
```

continues to check until mail is received.

One useful characteristic of the **MAIL** function is that like any decent mail system it will collect messages. Thus if several messages have collected from one or more sources, the **MAIL** function will deliver the oldest undelivered message and keep the others for future reference. A **SEND 255** goes onto every turtle's list. That message disappears from all lists when one turtle accepts it.

Now that you have the whole set of procedures in, you can try running the game. To start it, run **CATCHEM**. Just remember that this is an educational experience, not pure entertainment. You may discover that there is a flaw in the game. If the runner is caught close to home, then, because the chaser is returned to home after each successful catch, the runner is unable to escape and the score mounts. You could fix this by moving the chaser elsewhere if the runner is too close to home, or by just incrementing the x position of the chaser by some large number (say 100) after each catch.

The second game is called **REBOUND**. It makes use of the game controllers. The object of the game is to bounce a ball off two turnable paddles onto a target. Here we'll need a few more turtles. We first assign four tasks: reading of the two controllers, a scorekeeper, and a trigger to start the whole thing off.

```
TO REBOUND
   CLEAR  HT
   HATCH 2 PADDLE1
   HATCH 3 PADDLE2
   HATCH 6 SCOREKEEP
   TRIGGER
END
```

Let's look at the paddle controls first.

```
TO PADDLE1
   HT  SX 60  SY 180
   TURN 0
END

TO TURN :P :X
   WHILE 1
      (MAKE :X PADDLE :P/2
       LINE 3
       SH 45 + 3*:X
       LINE 0
       WHILE PADDLE :P/2 = :X ()
      )
END

TO LINE :COLOR
   PC :COLOR
   FD 15  BK 15  BK 15  FD 15
END
```

PADDLE1 establishes the position of the first paddle on the screen. It calls **TURN** which actually reads the game controller. The new idea in **TURN** is the use of the **PADDLE** function. The **PADDLE** function returns a number between 0 and 63 for the designated input; the number depends on the position of the controller handle. The inputs are 0 and 1 for the horizontal and vertical positions of the left game controller and 2 and 3 for the horizontal and vertical positions of the right game controller. Thus **PADDLE1** by the instruction **TURN 0** tells the procedure **TURN** to read the horizontal position of the left controller (left refers to the position of the plug on the rear of the Color Computer). Because the instruction is

```
MAKE :X PADDLE :P/2
```

the variable **:X** is a number between 0 and 31. This division of the controller reading by 2 reduces the sensitivity of the display and speeds response. Notice that, after the first pass through **TURN**, the procedure looks for a change in the controller setting. It stays in the loop

```
WHILE PADDLE :P/2 = :X ()
```

until there is a change. When there is a change, it runs through the outer loop which updates **:X**, erases the old paddle (**LINE 3**), computes a new heading (**SH 45 + 3*:X**), and draws a new paddle (**LINE 0**). Remember that **:X** can be between 0 and 31, so the heading for the paddle can be between 45 and 45 + 3*(31) = 138. The procedure **LINE** actually draws the paddles and erases them. The **BK** is broken into two steps so that it exactly duplicates the **FD** steps; this insures a successful erase.

The second paddle is controlled by the second controller. We can use **TURN** and **LINE** again.

```
TO PADDLE2
   HT  SX 180  SY 12
   TURN 2
END
```

Now we have to create the ball and the target. **TRIGGER** starts a new round.

```
TO TRIGGER
   HT
   HATCH 4 BALL
   VANISH
END
```

The ball should come from a randomly selected point towards the first paddle. The easiest way to do that is to create the ball turtle at the first paddle and to move it (invisibly) in the randomly selected direction. These two tasks will be carried out by the procedures **LAUNCHBALL** and **STARTSPOT**.

```
TO BALL
   LAUNCHBALL
   WHILE MAIL 5 = 0
      (STARTSPOT
       HATCH 5 TARGET
       REPEAT 45
         (FD 10
          IF NEAR 2<20
            (FD 10  LT  (HEADING 4 -
             HEADING 2 + 180)*2  FD 35)
          IF NEAR 3<25
            (FD 10
             LT  (HEADING 4 -
             HEADING 3)*2  FD 45)
      )
   )
   TRIGGER
END
```

At the same time we create the target at a randomly selected position (**HATCH 5 TARGET**). The **REPEAT** loop actually moves the ball. If the ball is close to the first paddle (turtle 2), the heading of the ball is changed

```
LT  (HEADING 4  - HEADING  2 +180)*2
```

There is a similar change when the ball is close to the second paddle (turtle 3). Notice that when the ball has moved the maximum distance it triggers a new ball before disappearing.

```
TO LAUNCHBALL
   HT  PU
   SHAPE UFFRRDFRFRFFRFRFFRFRFFR-
FRF
   MAKE :Y RANDOM 60 + 160
END

TO STARTSPOT
   HT  SH :Y  SX 60  SY 180
   REPEAT 6  (FD 10)
   WHILE XLOC 4>7 & YLOC 4>7
      (FD 10)
   RT 180  ST  FD 10
END
```

LAUNCHBALL creates an appropriate shape for the ball and effectively picks a random starting point by picking the heading. **STARTSPOT** hides the ball turtle, locates it at the first paddle, moves it until it reaches the edge of the screen, and finally turns it around and makes it visible.

TARGET does the scoring. First it picks a random position and creates a target shape. Then it watches for a close approach of the ball from below (if the ball approaches from above, it has not bounced off the second paddle). If the ball (turtle 4) comes close enough, then a message is sent to the scorekeeper and to the ball.

```
TO TARGET
   SH 0
   HT  SX RANDOM 100 + 135
   SY RANDOM 40 + 120
   SHAPE URRFFFFFFFFFLLLDFFFF-
FLFFFFFFFFFLFFFFF
   ST
   REPEAT 100
      (IF NEAR 4<15 &
       ABS  (HEADING 4 -180)>90
         (SEND 6 1  SEND 4 1)
      )
   VANISH
END
```

The **SCOREKEEP** procedure is essentially the same as before.

```
TO SCOREKEEP :SCORE
  HT  SX 200  SY 180
  WHILE 1
    (PRINT "  " PRINT :SCORE
     WHILE MAIL 5=0 ()
     MAKE :SCORE :SCORE+1
     COLORSET 1  COLORSET 0
  )
END
```

These two examples should help you to implement your own ideas for more complex games.

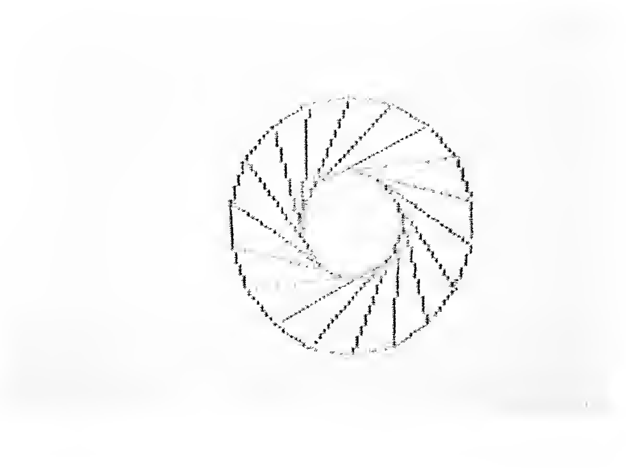
17. GRAB BAG

In this last chapter we give a final set of sample programs which we hope will give you ideas for your own projects. We have introduced all the features of Color LOGO earlier, so we will give these without lengthy comments.

The first set is controlled by the procedure **BOND**.

```
TO BOND
  WHILE 1
    (COLORSET 1
     CLEAR  HT  DELAY 1000
     TUNNEL
     WALK
     PAINT)
END

TO WALK
  SX 28  MAN2  ST  DELAY 2000
  REPEAT 29
    (MAN2  DELAY 100
     HT  SX XLOC ME + 3
     MAN1  ST  DELAY 100
    )
MAN2
DELAY 800  SX XLOC ME - 8
DELAY 500  SX XLOC ME + 16
DELAY 500  SX XLOC ME - 16
DELAY 500  SX XLOC ME + 8
REPEAT 3 (
  HT  DELAY 20
  ST  DELAY 30)
END

TO TUNNEL
  PC 1  HT  SX 60  SH 0
  REPEAT 18
    (FD 20  RT 124  FD 56
     BK 56  LT 104)
END

TO MAN1
SHAPE RRUFFFLLDFLFR-
FFLFFRRRFLLFFRRF-
LFLLLFFRRFLFRRFL-
FFLFLFLFLFFLFRFF-
FFLLFRRRFLFFRFL-
FFRRFF
END
```

```
TO MAN2
SHAPE RRUFFFFLLDFF-
FFLFFRRRFLLFFRRF-
LFLLLFFRRFLFRRFL-
FFLFLFLFLFFLFRFF-
FFLLFRRRFLFFFFFF
END

TO PAINT
   PC 2  HT  MAKE :X 1
   REPEAT 3  (COLORSET 0
      DELAY 100  COLORSET 1
      DELAY 100)
   SX 114  SY 102  SH 0
   REPEAT 13
      (RAGGED :X
       SX XLOC ME - 6
       SY YLOC ME - 2
       MAKE :X :X + 5
   )
END

TO RAGGED :X
   REPEAT 8
      (FD :X  RT 135  FD 8
       BK 8  LT 90)
END

TO DELAY :TIME
   REPEAT :TIME ()
END
```



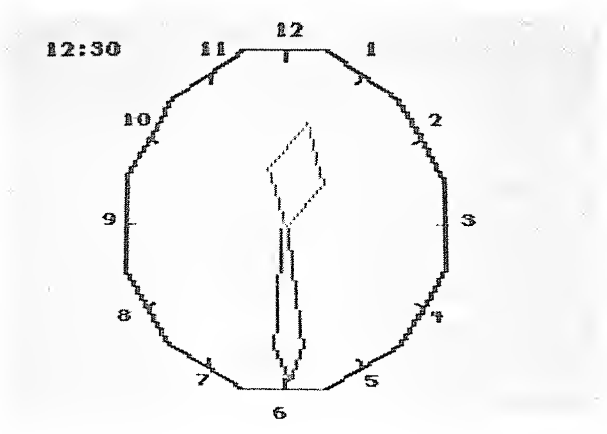
The next set is for a younger audience.

```
TO CLOCK :DELAY :INT
  CLEAR
  CLOCKFACE
  TIME :DELAY :INT
END

TO CLOCKFACE
  MAKE :NUMBER 12
  SY 180  SX 104  SH 90
  REPEAT 12
    (FD 22  RT 90  FD 5  BK 5
     PU  BK 10  PRINT :NUMBER
     FD 10  PD  LT 90  FD 22
     RT 30
     MAKE :NUMBER  :NUMBER+1
     IF :NUMBER > 12
       (MAKE :NUMBER 1) )
END

TO TIME :DELAY :INTERVAL
  HT
  REPEAT 24
    (MAKE :HR 0
     WHILE :HR<12
       (MAKE :MIN 0
        WHILE :MIN<60
          (DIGITAL :HR :MIN
           PC 1  LITTLEHAND :HR :MIN
           PC 2  BIGHAND :MIN
           REPEAT :DELAY ()
           PC 3  LITTLEHAND :HR :MIN
           BIGHAND :MIN
           MAKE :MIN
                   :MIN + :INTERVAL)
       MAKE :HR  :HR + 1) )
END

TO BIGHAND :MINUTE
  SX 128  SY 96  SH 6*:MINUTE
  LT 8  FD 60  RT 30  FD 18
  RT 130  FD 18  RT 32  FD 60
END
```

```
TO LITTLEHAND :HOUR :MINUTE
   SX 128  SY 96
   SH 30*:HOUR + :MINUTE/2
   LT 32  FD 30  RT 60  FD 30
   RT 120  FD 30  RT 60  FD 30
END

TO DIGITAL :HOUR :MINUTES
   SX 0  SY 180  PRINT "        "
   SX 8*(:HOUR<=9 & :HOUR<>0)
   IF :HOUR  (PRINT :HOUR)
   ELSE  (PRINT 12)
   SX 16  PRINT ":"  SX 24
   IF :MINUTES<10
     (PRINT "0"  SX 32)
   PRINT :MINUTES
END
```

Notice that you can set the interval to any number of clock minutes and that you can set the speed with **:DELAY**. Try running

```
CLOCK 300 5
```

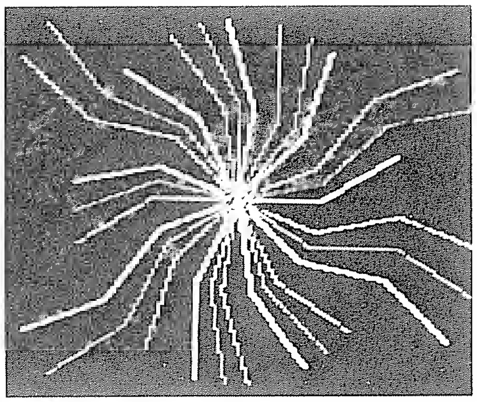


Next we give another colorful design.

```
TO SPIDER :X
   COLORSET 1  BG 0
   REPEAT 36
     (HATCH 1  OFFSET :X :C
      MAKE :C  :C+1  RT 10)
   VANISH
END

TO OFFSET :LENGTH :COLOR
   PC :COLOR  FD :LENGTH
   LT 30  FD :LENGTH
   RT 30  FD :LENGTH
END
```

Try this with

```
SPIDER 45
```



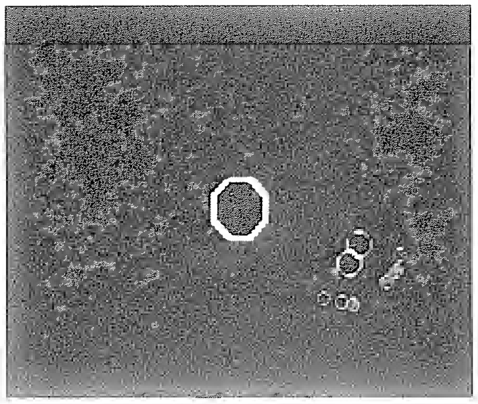
Next we give one which will remind you of the start of every science fiction film you've ever seen. There is no picture in the manual for this one, as the effect is all in the motion.

```
TO SPACETRAVEL
   COLORSET 1  BG Ø  HT
   MAKE :X 4
   WHILE 1
      (HATCH 1  STAR1
       RT 67
       HATCH 1  STAR2
       RT 2Ø7
       HATCH 1  STAR1
       RT 114
       HATCH 1  STAR2
       RT 87
       SETX XLOC ME + :X
       IF NEAR 255>3Ø
         (MAKE :X :X*-1
          HATCH 1 PLANET)
      )
   VANISH
END

TO STAR1
   HT
   SHAPE FFRRFRRF
   PU  FD 2  ST
   REPEAT 25  (FD 3)
END
```

```
TO STAR2
   HT
   SHAPE F
   PU  FD 2  ST
   REPEAT 35  (FD 3)
END

TO PLANET
   HT
   IF XLOC ME>128  (SETH 75)
   ELSE  (SETH 300)
   FD 10
   SHAPE FFRFFRFFRFFRFFR-
FFRFFRFF
   PU  FD 6  ST
   REPEAT 20  (FD 4)
END
```

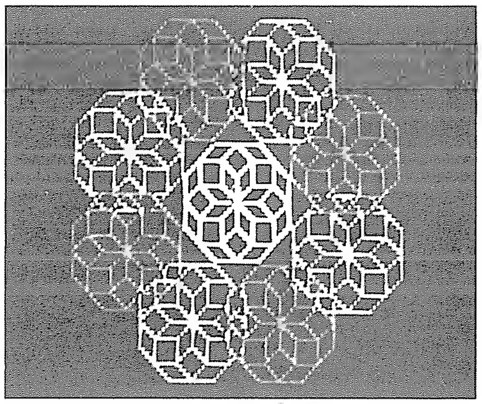
Here's one which shows the orbit of a moon around a planet and which makes use of multiple turtles to simplify the mathematics.

```
TO ORBIT
   COLORSET 1  BG 0
   FD 10  RT 90  PC 3
   REPEAT 8  (FD 6  RT 45  FD 6)
   HOME
   PU  SETH 90  SY 164
   MAKE :MOONPOS 0
   SHAPE U-
FFFFFRRDFFRFFFFRFFFFRFFFF-
RFFFFRFFFFRFFFFRFFFFRF
   WHILE 1
      (REPEAT 4
         (HATCH 1  MOON :MOONPOS
          REPEAT 6 ()
          MAKE :MOONPOS  :MOONPOS + 20
      )
      FD 10  RT 9
   )
END

TO MOON :POS
   HT  PU  RT :MOONPOS
   FD 20
   SHAPE UFFFFRRDFRFFRFFRFFR-
FFRFFRFFRFFRF
   ST
   REPEAT 9 ()
   VANISH
END
```



As our last example we give a final pretty pattern.

```
TO SAMPLE
   COLORSET 1  BG Ø
   NPOLY 8  12  3
   SX 7Ø  SY 72
   N2POLY  8  48  12
END

TO NPOLY :N :S :C
   PC :C
   REPEAT :N
      (POLYGON :N :S
       RT 36Ø/:N)
END

TO POLYGON :N :S
   REPEAT :N  (FD :S  RT 36Ø/:N)
END

TO N2POLY :N :S1 :S2 :I
   HT  PU  MAKE :I 1
   WHILE :I<=:N
      (HATCH :I NPOLY :N :S2
                  (1+:I-:I/2*2)
       FD :S1  RT 36Ø/:N
       MAKE :I  :I+1
      )
   VANISH
END
```



Of course you can try this set with other inputs than those given in **SAMPLE**.

Well, we have now reached the point where you are on your own. We are sure that the examples herein have just scratched the surface of what is possible. We hope that you have as much enjoyment working out your own demonstration procedures as we have had in developing these.

APPENDIX 1

LANGUAGE SUMMARY OF MICROPI COLOR LOGO FOR THE RADIO SHACK COLOR COMPUTER

STARTING LOGO

From Plug-in ROM:	Plug the Color LOGO cartridge into the game slot, then turn on the computer.
From Disk:	Put disk in drive Ø, type **LOADM "LOGO"** and press the **ENTER** key, wait for the drive light to go out, type **EXEC** and press **ENTER**.

MODES IN COLOR LOGO

The Color LOGO system can be in one of 4 "modes" depending upon what the user is doing at the time. A brief explanation of each is given here.

BREAK MODE	is entered upon system startup and by pressing **BREAK** at any time. In this mode the user can load and/or save programs from tape or diskette, make printed copies of programs or get into EDIT or RUN modes.
EDIT MODE	is entered from BREAK MODE by pressing the **E** key. In this mode the user can view, create or modify programs.
RUN MODE	is entered from BREAK MODE by pressing the **R** key. In this mode the user can enter turtle commands, call programs to be run, or enter DOODLE MODE.
DOODLE MODE	is entered from RUN MODE by pressing the **@** key. In this mode the user can doodle a picture while creating a procedure using specially marked keys.

BREAK MODE

BREAK MODE is entered automatically upon starting Color LOGO, and can be entered from any other mode by pressing the **BREAK** key at any time.

It is signified by the "**LOGO**:" prompt on the screen.

The following commands may be used in BREAK mode.

SHIFT CLEAR	clears the internal program area.
R	enters RUN MODE.
E	enters EDIT MODE.
P	prints contents of internal program area on the printer connected to the serial port.
Q	prints same as **P** command except the **Q** sends a line feed after a return character.
L	prompts for module designation with "**LOAD:**," then reads from the specified source into the internal program area. To load from tape enter "**T**." To load from disk enter the module identifier, which is a single letter from A to P. Both require an ENTER to start. The disk version allows program loading from disk or tape. The ROM version loads programs only from tape.
S	prompts for the module name with "**SAVE:**," then writes the internal program area to the specified destination. To save on tape enter "**T**," to save on disk enter the module identifier, which is a letter from A to P. Both require an ENTER to start. The disk version allows program saving to disk or tape. The ROM version saves programs only on tape.

EDIT MODE

EDIT MODE is entered from BREAK MODE by pressing E. In EDIT MODE one can edit the currently loaded modules. To start with a blank program area, press SHIFT CLEAR in BREAK MODE before pressing E.

The editor is very easy to use. It works on the principle that "what you see is what you get." The first line of text (if there is one) is displayed on the bottom line. To enter lines of text just type them on the screen. The cursor will always appear on the bottom line, but the text may be moved up or down the screen at will. The following keys cause special actions to take place.

ENTER	moves the text up one line on the screen, or if already on last line, then adds a new line to the text end.
↑	moves the text up one line unless already on the last line.
↓	moves the text down one line unless already on the first line.

←	moves the cursor left one character unless already at the beginning of line.
→	moves the cursor right one character unless at the line end.
CLEAR	moves to the top line of the text.
SHIFT ↑	scrolls the text up continuously until any key is pressed.
SHIFT ↓	inserts a blank line in front of the current line if the cursor is in column 1 (the current line bumps down off the screen); if not in column 1, then the current line is split at the cursor location into two lines.
SHIFT ←	deletes the character under the cursor and moves the remainder of the line left to close the gap. If the line has no characters, then the blank line is removed.
SHIFT →	inserts a blank into the line at the cursor location by moving the remainder of the line right one space. If the line is already full, then no action takes place.
BREAK	exits EDIT MODE and returns to BREAK MODE.
@	allows the next character to be one of the specially marked single key command codes. To enter a real "**@**" press @ twice.

In general, to enter new lines just type each line followed by an ENTER press. To modify a line, move the cursor into place with the arrow keys, then modify text by typing the new text over the old or by inserting or deleting characters as described above.

NOTE: If the editor quits accepting new text, then the program area is full.

The editor is general enough to be used not only for writing Color LOGO programs but for simple word processing applications. After you edit a text file you may print or save it on disk or cassette for later use. One such use would be writing documentation for modules written in Color LOGO. Since the editor has a maximum line length of 32 characters, a facility is provided to allow for printing of longer text lines on the printer. If a line is ended with an "**@**" character, then no **RETURN** is output at the end of the line. The result will be that the following line on the screen will be printed on the same printer line.

INTERNAL PROGRAM AREA

Color LOGO procedures are entered in the EDIT mode. They can then be saved on disk or tape and re-loaded later to be run again. The program area can have any number of Color LOGO procedures in it. Each procedure begins with a "**TO**" statement. The "**TO**" statement must be the first and only statement on a line. Other than that, any number of statements can share a line; each one is separated from the previous one by one or more spaces. Each procedure should end with an "**END**" statement. The work area may contain many procedures at once. It is a good practice to leave at least one blank line between procedures to improve readability. It is also suggested that program lines be indented to show the logical structure of the program. The examples in this manual are all written in this manner.

TURTLE SPACE

The TURTLE is a creature that has a visible SHAPE, a POSITION and a HEADING. The position is defined by an (X,Y) coordinate pair. The heading is defined by an angle from Ø to 359. In general, the turtle lives on the plane of the display screen. Using TURTLE GRAPHICS COMMANDS, you can make the turtle move about and, if desired, leave a trail. Initially a turtle starts at the HOME position. The home position is the approximate middle of the screen (X = 128, Y = 96). The turtle heading at HOME is zero degrees or straight up. The screen dimension in the X direction (across the screen) goes from Ø at the left edge to 255 at the right edge. The screen dimension in the Y direction (up and down) goes from Ø at the left edge to 255 at the right edge. The screen dimension in the Y direction (up and down) goes from Ø at the bottom to 191 at the top. The lower left hand corner of the screen has coordinates (Ø,Ø). The upper right corner has coordinates (255,191). The screen is normally a wrap-around space; that is, if the turtle runs off the top of the screen it appears on the bottom. If it runs off the left it appears on the right, etc. In that sense, the plane on which the turtle walks is infinite in any direction. The turtle may be pointed in any direction from Ø to 359 degrees. Straight up is Ø degrees, and the direction increases as the turtle rotates to the right, or in a clockwise direction.

RUN MODE

RUN MODE is entered from the BREAK MODE by entering $\boxed{\text{R}}$. When RUN MODE is entered, the screen is cleared, and the turtle appears at the bottom position. A TEXT WINDOW of three lines exists at the bottom of the screen. The user enters turtle graphics commands or calls Color LOGO procedures that have been entered earlier via the EDIT or DOODLE MODE. The user can enter any of the following commands directly in RUN MODE. However, in RUN MODE no more than one command may be entered on a line. Once the $\boxed{\text{ENTER}}$ key is pressed, the command is executed.

COMMANDS WHICH CAN BE ENTERED DIRECTLY IN RUN MODE

CLEAR	**HOME**	**FORWARD**	**BACK**
RIGHT	**LEFT**	**PENUP**	**PENDOWN**
PENCOLOR	**SHOWTURTLE**	**HIDETURTLE**	**SETX**
SETY	**SETHEADING**	**SLOW**	**COLORSET**
BACKGROUND	**WRAP**	**NOWRAP**	**SEND**
PRINT	**HATCH**	**VANISH**	

Some of these commands may be abbreviated as shown below. Other Color LOGO commands may not be entered in RUN MODE; they may only be used within a Color LOGO procedure.

TO EXECUTE A COLOR LOGO PROCEDURE FROM RUN MODE

To run a procedure entered earlier via EDIT or DOODLE MODE, just enter the name of the procedure. Follow the procedure name with any arguments to be passed to the procedure, then press $\boxed{\text{ENTER}}$. Each argument is preceded by at least one space. An argument can be a number, a variable or an expression. If an expression is used it must be enclosed in parentheses.

DOODLE MODE

DOODLE MODE is entered from RUN MODE by pressing the $\boxed{@}$ key. DOODLE MODE allows the creation of a turtle graphics procedure that will draw a shape without requiring that the user even know how to read. In DOODLE MODE the screen displays an "=" sign. The user enters a word (nonsense or otherwise) of at least one letter or number and presses $\boxed{\text{ENTER}}$. The word is the name of the procedure to be created as a picture is drawn. Now the numeric keys (marked by the special keyboard overlay) can be used to enter turtle graphics commands. Each time a key is pressed, the specified command is executed by the turtle. At the same time, a procedure is created in the program area. This procedure can be viewed by entering EDIT MODE. When entering commands, the BACKSPACE key can be used to erase the last command. In this case, the entire screen is erased and the shape is re-drawn without the last entered command. To exit the DOODLE MODE press $\boxed{\text{BREAK}}$. A procedure created in DOODLE MODE can be called out from RUN MODE to re-draw the picture again. To do so, just enter the name that was given when DOODLE MODE was entered.

The DOODLE MODE commands are:

(1) **CLEAR**	(2) **HOME**	(3) **PU**	(4) **PD**	(5) **RT 45**
(6) **LT 45**	(7) **FD 1**	(8) **FD 1Ø**	(9) **RT 15**	(Ø) **LT 15**

COLOR LOGO STATEMENTS AND COMMANDS

CONTROL STATEMENTS

IF expr (list of statements)	The expression is evaluated. If the value is true (non-zero) the list of statements in parentheses is executed. If it is false (Ø) then the list of statements is skipped. The word **THEN** may be inserted after the expression if desired. The **IF** statement may be followed by an **ELSE** statement. The "list of statements" denoted here and below (see **ELSE**, **REPEAT** and **WHILE**) consists of zero or more statements enclosed in parentheses. The statements may include any turtle commands or other control statements except the **TO** statement. There may be multiple statements per line and any number of lines may be used.
ELSE (list of statements)	This statement must follow an **IF** statement. If the expression value on the **IF** statement is false, then the list of statements after **ELSE** is executed. Otherwise it is skipped.
REPEAT expr (list of statements)	The expression is evaluated; if it has a value less than or equal to zero, then the list of statements is skipped. Otherwise the list of statements is executed the specified number of times.
WHILE expr (list of statements)	The expression is evaluated; if it is false (Ø), then the list of statements is skipped. If it is true (non-zero), then the list of statements is executed. After the list is executed, control returns to the **WHILE** again. The expression is then evaluated again. The list of statements is executed repeatedly until the expression is found to be false.
STOP	This terminates the execution of a procedure. Control is returned to the calling procedure if there is one. If the procedure was called from RUN MODE, then control returns to RUN MODE. If the procedure was called by a **HATCH** statement, then the associated turtle goes out of existence.
TO procname parmlist	This statement defines the start of a Color LOGO procedure. It must start in column 1 of a line and must be the only statement on the line. The "procname" may be any name of one or more letters. The parameters in the "parmlist" may be from Ø to 5 variables. Each one consists of a colon ":" followed by any word of one or more letters.

END	This is the last statement in a procedure. Execution of an **END** is equivalent to that of the **STOP** statement.
VANISH	**VANISH** takes the current turtle out of existence.
MAKE :var expr	The value of the expression is assigned to the variable.
PRINT "text" **PRINT** expr	The literal text or the expression value is displayed at the turtle location. The turtle is not moved.
NOWRAP	Normally, the screen is in "wrap" mode. That is, a turtle which runs off the screen will come back on the opposite edge. Execution of the **NOWRAP** statement takes the screen out of wrap mode. If a turtle then runs off the screen, the program will terminate with an "**OUT OF BOUNDS**" error message.
WRAP	Puts the screen back in wrap mode.
HATCH expr procname arglist	Creates a new turtle. The turtle will start at the same (X,Y) position as its parent (the turtle that HATCHed it) and will be pointed in the same direction. It will have the standard turtle shape. The expression value becomes the new turtle's identification number (a number from 1 to 254). The "procname" specifies the procedure to be executed by the new turtle. The "arglist" is optional; it specifies the arguments to be passed to the procedure. The new turtle runs simultaneously with the other active turtles.
SEND expr expr	A message is sent to the specified turtle. The first expression value denotes the identification of the turtle to which the message is sent. A value of 255 denotes that the message is being sent to the first turtle that requests its mail. Any other value denotes that the message can be received only by a turtle with the specified identification (see also the **MAIL** function). The value of the second expression is the value sent to the other turtle.

procname arglist	This is referred to as a CALL statement, even though it does not contain the word CALL. To CALL any procedure, just enter its name followed by any arguments to be passed. If arguments are present, they are separated by one or more spaces. Each argument may be a number, a variable, a function reference or an expression contained in parentheses. The argument's values are passed to the parameter variables on the **TO** statement of the called procedure. If there are fewer arguments than there are parameters on the **TO**, then extra parameters are set to Ø. If the called procedure executes a **STOP** or **END**, then control continues with the next statement after the call statement.
SLOW expr	The **SLOW** statement causes execution to slow down so that it can be watched more closely. The value of the expression denotes how slow to go. A value of 127 is the slowest speed. A value of Ø is full speed.

TURTLE GRAPHICS COMMANDS

Statement	Abbreviation	Meaning
BACK expr	**BK**	moves the turtle backward the number of steps denoted by the value of the expression. If the turtle's pen is down, then a line of the current pen color is drawn as the turtle moves.
BACKGROUND expr	**BG**	sets the background color of the screen to color Ø, 1, 2 or 3. The default background color is 3.
COLORSET expr		selects color set Ø or 1. For each set there are four distinct colors. The default colorset is Ø.
CLEAR		paints the entire display area the background color and moves the current turtle to the home position.
FORWARD expr	**FD**	moves the turtle forward the number of steps denoted by the value of the expression. If the turtle's pen is down, then a line of the current pen color is drawn as the turtle moves.
HIDETURTLE	**HT**	makes the turtle invisible.
HOME		sends the current turtle to position (128,96) with heading Ø.

LEFT expr	**LT**	turns the turtle left (counter-clockwise) the specified number of degrees.
PENCOLOR expr	**PC**	sets the pen color of the current turtle to color Ø, 1, 2 or 3. The default color is Ø. The actual color depends on the current color set. If the pen color is set to the same color as the screen background color, then the turtle pen will erase as it moves.
PENDOWN	**PD**	tells the current turtle to draw a line as it moves in response to **FORWARD** or **BACK** commands.
RIGHT expr	**RT**	turns the turtle right (clockwise) the specified number of degrees.
SETHEADING expr	**SETH** and **SH**	points the turtle in the direction specified by the expression. The heading can be from Ø to 359 degrees. Ø degrees is straight up.
SETX expr	**SX**	moves the turtle by changing its X coordinate to the value specified. No line is drawn. The value may be from Ø (left edge) to 255 (right edge).
SETY expr	**SY**	moves the turtle by changing the Y coordinate to the value specified. No line is drawn. The value may be from Ø (bottom) to 191 (top).
SHAPE shape list		changes the shape of the current turtle to a shape denoted by the shape list. See TURTLE SHAPE LIST below.
SHOWTURTLE	**ST**	makes the turtle visible.

EXPRESSIONS

The "expr" designation above denotes a place in which an expression can be substituted. An expression can be a number, a variable, a function reference or a combination of these and the operators shown below. An expression is always evaluated to a number from -32768 to 32767.

Expressions may contain parentheses and are, in general, evaluated in the same manner as those in BASIC or other languages.

ARITHMETIC OPERATORS

These operators result in a number from -32768 to 32767.

+	addition	−	subtraction
*	multiplication	/	division

LOGICAL AND RELATIONAL OPERATORS

These operators always result in a 1 for TRUE or a 0 for FALSE.

&	logical AND	!	logical OR
NOT	logical negation		
<	less than	>	greater than
=	equal to	<>	not equal to
<=	less than or equal to	>=	greater than or equal to

VARIABLES

A variable consists of a colon followed by a word containing any number of letters or numbers. A variable represents a unique storage location for a number. If a variable is given on a **TO** statement, then that variable is said to be a LOCAL variable. That is, each time the procedure is invoked, a new storage location is assigned to the variable. Thus, if a procedure is invoked recursively or by several turtles at once, then each invocation has its own set of local variables which, though they have the same name, are kept distinct. There may be up to 5 parameters on a **TO** statement; thus there may be up to 5 local variables in a procedure.

If a variable is referenced in a procedure but is not on the **TO** statement for the procedure, then the variable is said to be a GLOBAL variable. There is only one storage location assigned to each particular global variable. Thus all references to the global variable refer to the same storage location even if the references are in different procedures. This provides a way of sharing information among procedures or among turtles.

LITERALS

'C	A quote (') followed by *one character* is called a *literal*. It can be used anywhere a number can be used. The value of a literal is the ASCII value of the character. For example, **'A** is equal to 65. A literal is particularly useful in checking for values returned by the **KEY** function.

FUNCTIONS

ME	returns the identification of the current turtle. The main turtle is number Ø. The others are numbered from 1 to 254.
KEY	returns Ø if no key is depressed. If a key is depressed, then the value is the ASCII value of the character.
ABS arg	returns the absolute (positive) value of the argument.
RANDOM arg	returns a random number from Ø to arg-1.
XLOC arg	returns the X coordinate of the turtle with the specified identification (note "**XLOC ME**" gives your own X coordinate). If no turtle exists with the identification, then 128 is returned.
YLOC arg	returns the Y coordinate of the turtle with the specified identification (note "**YLOC ME**" gives your own Y coordinate). If no turtle exists with the identification, then 92 is returned.
HEADING arg	returns the heading (Ø to 359) of the turtle with the specified identification (note "**HEADING ME**" gives your own direction). If no turtle exists with the identification, then Ø is returned.
NEAR arg	returns a measure of the distance from the current turtle to the one with the specified identification. The measure is equal to the number of steps in the X direction plus the number of steps in the Y direction. If no turtle exists with the specified identification, then the distance to **HOME** is measured.
MAIL arg	returns a number value. **MAIL** is used to check for and receive messages sent via the **SEND** command. The argument of the **MAIL** function denotes the source from which messages are to be received. If the argument is 255, then mail is received from any turtle that has sent mail addressed to the current turtle. If the argument is not 255 then it denotes the identification of the turtle from which mail is to be received. If more than one message is available for delivery, then the oldest undelivered message is the one returned. If no messages are available, then a value of zero is returned.

PADDLE arg	returns a value from Ø to 63 denoting the position of one of the game paddles (joysticks). The arg is a value from Ø to 3. **PADDLE Ø** gives the UP/DOWN of the left paddle. **PADDLE 1** gives the RIGHT/LEFT of the left paddle. **PADDLE 2** gives the UP/DOWN of the right paddle. And **PADDLE 3** gives the RIGHT/LEFT of the right paddle. For UP/DOWN the minimum value is UP. For RIGHT/LEFT the minimum value is LEFT.

The "arg" designation shown above denotes an argument value passed to a procedure or function. The argument can be a variable, a number, a function reference or an expression. If the argument is an expression, it must be enclosed in parentheses.

TURTLE SHAPE LIST

The **SHAPE** statement is used to assign a new shape to the current turtle. The shape of a turtle is made up of a pattern of dots on a grid. The shape list tells Color LOGO how to draw the turtle pattern. The turtle shape is automatically rotated to face in the direction the turtle is headed. Drawing the turtle shape is similar to using normal turtle graphics commands to draw any shape. The difference is that the commands which make up the shape list are a restricted and simplified form of the normal turtle graphics commands. The commands allow a step of one pixel (one square on a piece of graph paper) in any of the 8 possible directions. The 8 directions are: up, down, right, left, and the four diagonal directions. The one-letter commands that may be used in a shape list are shown below. The shape list can be any length. If it runs over a line boundary, put a hyphen "-" at the end of the line, then continue in column 1 of the next line. The turtle shape drawing pen complements the affected pixels. That is, the complement of "a dot present" is "no dot present" and vice versa. This allows a turtle to pass over a picture without destroying the picture.

TURTLE SHAPE COMMAND	MEANING
F	step forward one pixel; if the pen is down, complement the pixel.
B	step backward one pixel; if the pen is down, complement the pixel.
R	rotate right by 45 degrees.
L	rotate left by 45 degrees.
U	pick up the turtle shape pen; this pen is always assumed down at the start of a shape list. This pen should not be confused with the turtle pen that draws when a **FORWARD** or **BACK** turtle graphics statement is executed.
D	put the turtle shape pen down; if the pen was previously up, then putting it down will cause the current pixel to be complemented.

MULTIPLE TURTLES

Normally one turtle exists. The user can create additional turtles by using the **HATCH** statement. Each turtle then runs its procedures independently of the other turtles. The **HATCH** statement assigns an identification number to each turtle. That number may be used by other turtles to send mail or request location information about the turtle. The main turtle is always number Ø. Other turtles can have a number from 1 to 254.

When a turtle other than the main one exists from the procedure given it when it was HATCHed, it goes out of existence leaving behind only the lines it drew on the screen. The **VANISH** statement also causes the turtle to go out of existence. The main turtle, in contrast, can only go out of existence by executing a **VANISH** statement.

If the main turtle exits from the procedure given to it from RUN MODE, it will return to RUN MODE where the user can then enter its next command or procedure to run. If when the main turtle is in RUN MODE there are other turtles, then the other turtles cease to move. Each time the **ENTER** key is pressed, each of the turtles executes one program statement. This has the effect of stepping the hatched turtles along at a controlled pace. A useful debugging method is to **HATCH** a turtle from RUN MODE and tell it to run the procedure which is to be tested. Then the procedure is run by pressing **ENTER** repeatedly. If you enter a **VANISH** command, then the main turtle will disappear and the hatched turtle will run at full speed.

ERROR MESSAGES FROM COLOR LOGO

In BREAK MODE a "?" is displayed if any key other than a valid command letter is pressed. A load or save command may also display a digit followed by a "?".

1?	memory error
2?	tape checksum error (probably a bad tape or the volume not set correctly
3?	attempt to load a tape that is not a Color LOGO program
4?	attempt to load a module that is too long for memory
6?	disk drive not ready, or an attempt to write on a write-protected diskette, or an improperly formatted diskette.

In RUN MODE there are several possible messages that may be issued. These messages attempt to identify the error in the program, but remember that the message is only a "guess" as to what is wrong. It is possible that the message does not exactly fit the problem.

Each time one of the following messages is displayed, the user must press any key to continue.

MESSAGE	PROBABLE MEANING
I DON'T KNOW HOW TO ...	"..." is filled in with the name of what Color LOGO thought was a procedure name to call; but the procedure name is not found in the program area. If the name is one which should be in the program area, make sure that it is preceded by "**TO**" (not "**TØ**" (zero)). Also make sure "**TO**" is in column 1. Check also that the name is correctly spelled. If the name was not supposed to be a procedure, then probably there is something wrong with the immediately preceding command.
I CAN'T FIGURE OUT ...	"..." is filled in with the word that caused the confusion. Color LOGO was attempting to compute the value of an expression when it encountered the problem. Possibly the syntax of the expression is in error; or a colon is left out before a variable name; or a function name is misspelled.
I DON'T KNOW HOW MUCH	This message means that a command such as **RIGHT** or **FORWARD** which should be followed by a number is not followed by a number. Either an expression is not present where one should be or the very first item in the expression is not valid.

"(" OR ")" NOT RIGHT	A left parenthesis is not found as expected after an **IF**, **WHILE** or **REPEAT** expression or after an **ELSE**. Or, unbalanced parentheses are detected.
I CAN'T DO THAT IN THIS MODE	A command (such as **REPEAT**) other than one of the ones allowed is entered directly from the keyboard in RUN MODE. Remember, some commands may be executed only within a Color LOGO procedure.
MY MEMORY IS TOO FULL	The internal program and work area is filled. This will always happen eventually if a program is allowed to do infinite recursion (call itself repeatedly forever). In general, procedure calls, hatching turtles and sending messages consume memory. The longer the text in the program area, the less available memory for these operations.
OUT OF BOUNDS	The screen has been placed in NOWRAP mode and a turtle has run off the boundaries of the screen.

FORMAT OF A COLOR LOGO DISK

A Color LOGO program disk is formatted with the "**DSKINIØ**" command under BASIC. However, Color LOGO does not use the BASIC directory. Color LOGO divides the diskette into 16 modules named A through P. Once a diskette is used under Color LOGO it should not be written upon under BASIC. Each Color LOGO module occupies 2 tracks or 36 sectors on the diskette.

APPENDIX 2

MAKING A BACKUP COPY OF THE COLOR LOGO DISKETTE

It is a good idea to make a copy of the Color LOGO diskette for everyday use. The original Color LOGO diskette can be stored in a safe place to protect it from damage. To make a backup copy, follow the steps below.

One-Drive TRS-80 Color Computer Disk System

1. Make sure that the Color Computer is properly connected to the color television or color video. Plug the Color Computer Disk Controller into the slot on the right side of the computer.

2. Turn the color video and the Color Computer system on. (The computer's power switch is on the back left corner of the computer. The disk drive switch is on the back of the drive, in the upper corner.)

3. When you see the "**OK**" prompt, insert a new, blank diskette into the disk drive. Then type D S K I N I Ø and press ENTER.

4. When the "**OK**" prompt reappears, remove the new diskette from the disk drive.

5. Place an adhesive tab (provided with new diskettes) over the square notch in the Color LOGO diskette. (If you do not have any tabs, use a small piece of opaque tape.)

6. Insert the Color LOGO diskette with the square notch up and the label facing right, into the disk drive. Close the disk drive latch.

7. Type B A C K U P ␣ Ø and press ENTER.

8. When you see the message, "**INSERT DESTINATION DISKETTE AND PRESS ENTER**", remove the Color LOGO diskette (called the "**SOURCE**" diskette) from the disk drive. Insert the new diskette that you used in Step 3 (the "**DESTINATION**" diskette) into the disk drive. Close the disk drive latch. Finally, press ENTER.

9. When you see the message, "**INSERT SOURCE DISKETTE AND PRESS ENTER**", remove the DESTINATION diskette from the disk drive, insert the Color LOGO diskette, close the disk drive latch, and press ENTER.

10. Continue to switch between the SOURCE diskette and the DESTINATION diskette as instructed by the computer. When the BACKUP process is complete, you'll see the "**OK**" prompt reappear.

Two-Drive TRS-80 Color Computer Disk System

1. Make sure that the Color Computer is properly connected to the color television or color video. Plug the Color Computer Disk Controller into the slot on the right side of the computer.

2. Turn the color video and the Color Computer system on. (The computer's power switch is on the back left corner of the computer. The disk drive switch is on the back of the drive, in the upper corner.)

3. Insert a new, blank diskette into Drive 1 (the disk drive second from the Color Computer on the cable). Close the disk drive latch.

4. Type **DSKINI1** and press **ENTER**.

5. Place an adhesive tab (provided with new diskettes) over the square notch in the Color LOGO program diskette. (If you do not have an adhesive tab, use a small piece of opaque tape.)

6. Insert the Color LOGO diskette in Drive Ø (the disk drive closest to the Color Computer on the cable). Close the disk drive latch.

7. Type **BACKUP Ø TO 1** and press **ENTER**.

8. The computer will copy the contents of the diskette in Drive Ø onto the diskette in Drive 1. When the BACKUP process is complete, you'll see the "**OK**" prompt reappear.

INDEX

Note that the language summary in the Appendix begins on page 107. Therefore, page numbers below 107 refer to discussions within the tutorial, and page numbers above 107 refer to summaries in the language reference appendix.

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U.K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

Printed in U.S.A.